A FORMAL METHODOLOGY AND TECHNIQUE FOR VERIFYING COMMUNICATION PROTOCOLS IN A MULTI-AGENT ENVIRONMENT

THESIS

Timothy H. Lacey, Captain, USAF

AFIT/GCS/ENG/00M-12

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DTIC QUALITY INSPECTED 4

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

A FORMAL METHODOLOGY AND TECHNIQUE

FOR VERIFYING CONVERSATIONS

IN A CLOSED MULTI-AGENT SYSTEM

Timothy H. Lacey, B.S.,
Captain, USAF
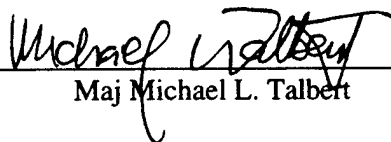
Approved:

_____
Maj Scott A. Deloach

_____
18 Feb 2000
date

_____
Dr. Thomas C. Hartrum

_____
18 Feb 2000
date

_____
Maj Michael L. Talbert

_____
22 Feb 2000
date

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my faculty advisor, Maj Scott Deloach, for his guidance and support throughout the course of this thesis effort. His insight and experience was certainly appreciated, and his teaching provided a wealth of knowledge that enabled me to complete this thesis. Thanks to Dr. Tom Hartrum for advice and guidance early on which greatly decreased the effort required to complete this research and to Maj Michael Talbert for serving on my Thesis committee. I would also like to thank my sponsor, Captain Freeman Alex Kilpatrick, from the Air Force Office of Scientific Research, for both the support and latitude provided to me in this endeavor.

Most importantly, I would like to express great appreciation to my wife Sandy, my sons Daniel, Joel, Joshua, and Jonathan, and my daughter Shauna, whose unwavering love, understanding, and sacrifice over the past 18 months allowed me to focus on my studies in this graduate program. Without their support, completion of this program would have been impossible.

Timothy H. Lacey

Table of Contents

TABLE OF FIGURES

ABSTRACT

As network bandwidth increases, distributed applications are becoming increasingly prevalent. Systems using these applications are very complicated to build and must be dependable. Software agents are ideal for breaking complicated problems into manageable subtasks. Agent conversations, a series of messages passed between agents, are the cornerstone of multi-agent systems and must be deemed correct before being placed into service. The purpose of this research was to develop a formal methodology and technique to verify that the communication protocols defined in a multi-agent environment were valid. This was accomplished by examining agent conversations before deploying the system. An additional goal of this research was to develop a proof-of-concept module for agentTool that automatically verified some of the important properties identified in this methodology.

A FORMAL METHODOLOGY AND TECHNIQUE

FOR VERIFYING CONVERSATIONS

IN A CLOSED MULTI-AGENT SYSTEM

I. Introduction

As network bandwidth increases, the Air Force is fielding increasingly distributed C^3I applications. This is clearly delineated by visionary documents such as *Joint Vision 2010* (Shalikashvili, 1999), and *Air Force 2025* (Kelley, 1996). The common thread in each of these documents is *information superiority*, which the Air Force believes will be the key factor to success in the 21st century. Distributed systems such as those required by the Air Force are very complicated to build, but must be dependable if the warfighters whose lives are at risk are going to trust them. Therefore, software engineers must ensure that the system and its information sources are robust, reliable, and secure. The Air Force's Office of Scientific Research is sponsoring research in intelligent software agents because they believe software agents are the appropriate mechanism for delivering these capabilities to the user. Distributed agents are well suited to applications that retrieve, filter, and summarize information as well as provide intelligent user interfaces and planning. The size and complexity of such a worldwide-distributed system will necessitate formal and rigorous approaches to ensuring the entire system will be interoperable and secure.

Before a multi-agent system can be trusted to perform as expected, the communication methods between the agents must be formally verified. The verification process includes checking for infinite loops, deadlocks, and other communication pitfalls that would prevent a

1

multi-agent system from completing its mission. This thesis designs and implements a methodology using formal methods that verifies that a system of agents will communicate as expected before a user deploys the system. Then, and only then, the user of the multi-agent system can be assured the system will communicate as expected.

1.1 Background

Many agent-based systems consist mainly of single agents. These agents do not have the capability to cooperate with other agents and jointly solve a problem. However, advances in technology and programming languages have enabled software engineers to create systems of multiple agents that "team up" to solve tough problems. It is apparent that agents in multi-agent systems have to communicate in a distributed environment and pool resources to solve problems.

The best way for software developers to tackle complex, large, or unpredictable domains is by breaking the problem into smaller, manageable tasks. Software agents can be used to solve these small tasks while working together to solve larger problems. Katia Sycara has observed that often agents must operate concurrently in a distributed environment to accomplish difficult tasks (Sycara, 1998).

1.2 agentTool

Agents communicate with each other using patterns of messages called *conversations*. Conversations may be structured and predictable, or they may be unstructured and dynamic. Structured conversations can be modeled using state transition diagrams. Given a set of conversation state transition diagrams, communication between agents can be simulated and all possible message combinations exercised. Using this approach, conversations are deemed *valid* if the desired message sequence takes place between the communicating agents. This process of deeming the conversations valid or invalid is called *verifying* the agent conversations.

Conversations can be verified manually (by a human analyst) or automatically (by intelligent software and automated tools).

The software development environment, agentTool, is being created at the Air Force Institute of Technology (AFIT) to address the need for a user friendly, robust tool for building multi-agent systems. The tool is designed to be an integrated environment that allows a user to graphically engineer a multi-agent system, verify the agent conversations with an automated verification tool, and automatically generate the source code for the designed system. This allows a user to specify a multi-agent system at three levels: domain, agent, and component. The *domain level* is where agent classes and interactions are defined. The *agent level* is where the internal agent architecture is defined. Lastly, the *component level* is where individual components in the system architecture are defined. During the domain level design, the communications between agents are specified as conversations. The system uses an automated verification tool and formal modeling languages to verify these conversations are valid. Feedback is provided to the user indicating whether the conversation design is valid. The automatic verification of agent conversations and message sequences using formal methods is the focus of this research effort.

The agentTool system incorporates the latest technology in multi-agent systems. A designer uses pre-defined or user-defined components while building an agent system and implements the system on various frameworks (Robinson, 2000). Users build agent systems with graphical analysis and design tools that are easy and intuitive to use (Wood, 2000). A knowledge base preserves agent designs and components providing agentTool with reusability, robustness, and extensibility (Rafael, 2000).

1.3 Problem Statement

Infinite loops, deadlocks, and other communication pitfalls can wreak havoc in a multi-agent system. Even worse, the system can appear to be working while an undetected catastrophic problem exists. The challenge is to explore paths that the conversation can feasibly encounter and formally verify the conversation is valid. Once the conversations have been verified, the user can trust the agents to communicate as expected.

Researchers at AFIT are currently developing agentTool. To ensure security and interoperability, agentTool must be able to enforce protocol policy on a proposed system. Therefore, the goal of this research is to *develop a formal methodology and technique to verify that the communication protocols defined in a multi-agent environment are valid. This is accomplished by examining agent conversations before deploying the system. An additional goal of this research is to develop a proof-of-concept module for agentTool that automatically verifies some of the important properties identified in this methodology.*

1.4 Assumptions

The following are assumptions concerning agentTool, designed agents, and their operating environment.

1) Agents designed in agentTool will be used in a *closed* environment. A closed environment is one in which all participants are known and all conversations are predetermined. An agent's behavior is predictable and agents communicate with each other via conversations.

2) Agents can assume more than one role at a time, and can be involved in multiple conversations at any given time.

3) Conversations can be started from within other conversations.

4) System variables can impact conversations in adverse ways. It is possible for an external factor to prevent a conversation from completing, even though the conversation is perfectly valid and has been verified. Therefore, it is assumed agentTool will not be able to detect errors caused by system variables while verifying conversations.

1.5 Thesis Overview

Chapter 2 provides a review of the relevant literature and research including formal languages, automated verification tools, and the types of agent conversation properties that can be verified. Chapter 3 specifies a methodology that takes a conversation specification and verifies it using an automated tool. Chapter 4 describes the application of the verification methodology and the prototype to agentTool. Finally, Chapter 5 presents conclusions and future work.

II. Background

2.1 Overview

This chapter reviews verifiable properties of agent conversations, and some of the languages and tools available for verifying properties of agent conversations. Section 2.2 explains how agents use conversations and how to model them. Why formal methods are needed to verify conversations is covered in Section 2.3. Section 2.4 describes properties of agents that are verifiable while providing simple examples of such properties. Finally, Section 2.5 presents three formal languages and corresponding automated verification tools for verifying agent conversations.

2.2 Multi-agent Systems and Agent Conversations

Agents in a multi-agent environment should communicate with each other with structured messages. This enables the users of the multi-agent system to have assurance agents will perform as designed without unpredictably performing some unassigned task autonomously. The structured sequence of messages is called an agent conversation. Granted, there are occasions when agents are used in *open* environments where they may encounter any type of agent. In an open environment, an agent must be able to dynamically construct its conversations. However, this research is concerned with *closed* environments where agents are aware of their surroundings and know who their fellow agents are. Perhaps most importantly, each agent knows how it is supposed to communicate with its fellow agents. Whenever an agent sends or receives a message, it passes through various states of a conversation. These states determine how the agent behaves.

An agent conversation consists of an *initiator* side and a *responder* side. Both sides of the conversation move through various states in harmony as the conversation develops. Eventually, both sides of the conversation will end up in its respective *end* state and the conversation will be completed. It is the state transition diagram that allows one to visualize the various states a conversation goes through and records the events that cause the conversation to move from state to state.

Figure 1 illustrates one side of a conversation and Figure 2 illustrates the complimentary side of the conversation. The two sides make up one complete conversation.
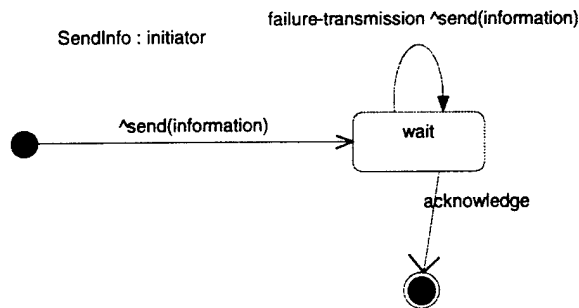
Figure 1: Initiator Half of Conversation SendInfo (DeLoach, 1999)
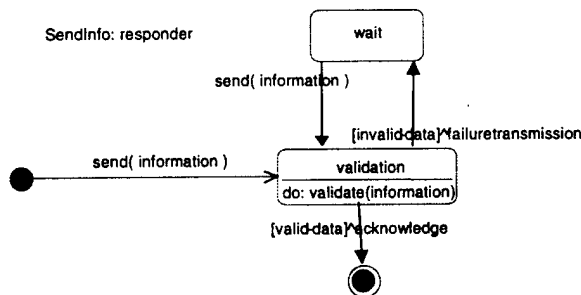
Figure 2: Responder Half of Conversation SendInfo (DeLoach, 1999)

The beginning state in a conversation is the "start" state. It is signified by a solid circle. The final state in a conversation is the "end" state and is signified by a solid circle with a ring drawn around it. Each intermediary state is drawn as an unfilled rounded edge rectangle. The state's name is inside the rectangle. Arrows between states indicate transitions between those

states and the direction of the transition. Labels on the arrows indicate the events and actions that take place to cause a transition from one state to another. The transition labels follow Unified Modeling Language (UML) notation. The labels are formatted as follows:

```
event-name(argument list)[guard condition]/action-expression^send-
                            clause
```

The label may contain some or all of this information. Each state may have more than one entry point and exit point, but all exit points must be deterministic. Referring to Figure 4, there are three states in the initiator side of the *SendInfo* conversation. They are the start, wait and end states. The transition from the start state to the wait state sends a send(information) message. The information is a parameter that is passed with the message. The transition from the wait state back to the wait state takes place when a failure-transmission message is received while in the wait state. This transition receives a failure-transmission message then sends a send(information) message before transitioning back to the wait state. Finally, the transition from the wait state to the end state takes place when an acknowledge message is received while in the wait state. No messages are sent during this transition and this side of the conversation ends.

2.3 Verifying Agent Conversations Using Formal Methods

Multi-agent software systems are difficult to build. Part of the research community believes multi-agent systems should be open ended and conversations between various agents should be dynamic and flexible (Sycara, 1998). Another part of the community believes agent conversations should be predetermined and structured so that all possible variants of a conversation are reproducible and verifiable (Harel, 1987). Some researchers have undertaken an effort to develop formal approaches to assist the software developer in the analysis and design of multi-agent systems (Holzmann, 1987). Fortunately, automated tool support is also available to

assist with formal methods. Many tools have been developed that analyze concurrent systems. These tools can also be used to verify agent conversations.

One of the simplest ways to verify agent conversations is with a technique called *reachability analysis* (Cleaveland, 1993). Automated tools are excellent for this technique. The first step in using an automated tool is to model the conversation using a language accepted by the tool. Some of the most popular languages to choose from are Communicating Sequential Processes (CSP) (Hoare, 1985), Calculus of Communicating Systems (CCS) (Milner, 1989), and Process Meta Language (Promela) (Holzmann, 1997). After modeling the proposed system using the required input language, the user may provide logical formulae describing undesirable states that the system should never reach. Given such formulae and the system description, the tool explores every possible state the conversation may reach during execution and checks to see if an undesirable state is reachable. If so, the automated tool reports a description of the execution sequence leading to the offending state. Using this approach, automated tools can find many undesirable conditions such as deadlock and critical section violations.

Reachability analysis falls under a more general type of verification called *model checking* (Cleaveland, 1993). Using this approach, an analyst describes a conversation using a design language, and then specifies properties the conversation should have as logical formulae. These formulae define behaviors the conversation should, or should not have as it executes and contains temporal operators enabling one to describe how a conversation behaves as time passes. Using such a temporal logic one can state properties such as the following:

- *The variable p will eventually become true*

- *It is mandated that after p becomes true, q will become true and remain true*

In the next section, an overview of communication system properties and the various methods of describing system properties are provided.

2.4 Properties of Communication Systems

Properties of communication systems in general fall under two broad categories: *generic* and *user-specified* properties. Generic properties that are applicable to all communication systems are deadlock, infinite overtaking, and livelock. User-specified properties of communication systems can be further broken down into *safety* and *liveness* properties.

2.4.1 Generic Properties of Communication Systems

2.4.1.1 Deadlock

A deadlock is a situation in which two computer programs sharing the same resource effectively prevent each other from accessing the resource, resulting in both programs blocked. When computer operating systems run only one program at a time all of the resources of the system are available to this one program. However, when operating systems run multiple programs at once, interleaving them with each other, programs can request resources dynamically. This can lead to the problem of deadlock. Here is a very simple example:

```
Program one requests resource A and receives it.
Program two requests resource B and receives it.
Program one requests resource B and waits for program two to
    release it.
Program two requests resource A and waits for program one to
    release it.
```

Now neither program can proceed until the other program releases a resource. The operating system has a dilemma and cannot know what action to take. At this point, the only alternative is to kill one of the programs. Learning how to handle deadlock situations has had a major impact on the development of not only operating systems but also communicating systems in general.

In agent conversations, deadlock can occur when both sides of a conversation wait to receive a message that never arrives. This dilemma can happen many ways. The message could be lost, an incorrect message could be sent, or the message could not be sent at all.

2.4.1.2 Infinite Overtaking

To demonstrate the concept of infinite overtaking, recall the infamous dining philosophers' example as portrayed by C.A.R. Hoare in his book, *Communicating Sequential Processes* (Hoare, 1985). A round table has been prepared with five chairs containing five philosophers and a bowl of pasta in the middle of the table. Each of the philosophers has a fork on the table between him and the other philosophers; thus, there are five forks in all. Before a philosopher can eat, he must have a fork in each hand. This means that not all five philosophers can eat at one time. Suppose a seated philosopher has a greedy left neighbor and a rather slow left arm. Before he can pick up his left fork, his left neighbor rushes in, sits down, quickly picks up his left and right forks, and has his fill of pasta. Eventually he puts down both forks and gets up to leave. Then the left neighbor gets hungry again, sits down, and quickly grabs both of his forks before his right neighbor has an opportunity to pick up the fork they share. Since the philosopher with the bottomless stomach can repeat this cycle forever, the seated philosopher to his right may starve to death.

One such agent-based scenario is where an agent requests information from a pool of information brokers. If one of the brokers happens to be extremely quick, the remaining brokers will never be able to answer any requests for information. The real loser in this scenario may be the requester, for he may only get information from one source.

11

2.4.1.3 Livelock

Livelock is a situation in which some critical stage of a task is unable to finish its processing. This is because the users of this particular task continuously create more work for the task to do after the critical stage of the task has provided the requested service for them but before the given task can clear its request queue. Livelock differs from deadlock in that the process is not blocked or waiting for anything but has a virtually infinite amount of work to do and can never catch up. An example of livelock is that of an interrupt driven operating system. If too many interrupts arrive at the operating system's kernel and then continue to bombard the kernel, the operating system will not be able to actually service any of the interrupt requests because it will spend all of its time processing the receiving of the interrupts. In other words, the operating system is so busy receiving interrupt requests it cannot service any of the requests.

Agent conversations can also succumb to livelock. A broker agent could be inundated with requests for information to the point where he could never respond to all the requests because his time is spent processing the receipt of requests.

2.4.2 User-defined Properties of Communication Systems

It is easy to take a snapshot of a system and analyze its properties. However, often it is more desirable to know if eventually something will happen or conversely, that something will never happen. This type of system property can be described using temporal operators. Many properties of agent conversations can be expressed with temporal operators. For example, we might want to know that if message A is sent to a recipient, eventually a reply will be received. This property can be stated as "it is always the case that eventually we receive a reply from the recipient." This is commonly known as message sequence verification.

Temporal logic is simply an extension of propositional logic. The difference between the two is that temporal logic has special operators that allow for time. Amir Pnueli defines a temporal operator called the *henceforth* operator [] (Manna, 1992). An example of how this operator would be used is []p, read henceforth p or always p. Therefore, []p holds at position q if and only if p holds at position q and all of the following positions from now until eternity.

Pnueli also defines a temporal operator called the *eventually* operator <>. An example of this operator is <>p, read as eventually p. Therefore, <>p holds at position j if and only if p becomes true at some position q where q>=j.

The combination of temporal operators can be used to form many types of user-defined properties. The next few sections accent the types of conversation properties that can be expressed with temporal operators.

2.4.2.1 Safety Properties

Safety properties have the form "bad things will not happen." These properties are expressed by logical statements that the system state must satisfy at all times as well as pre and post conditions. Preconditions reflect the state of a program before the execution of a set of statements. Postconditions reflect the state of a program after the termination of a set of statements.

2.4.2.1.1 Nontermination

As an example of a user-defined *safety* property, consider the property of nontermination of a conversation. A conversation is nonterminating if it never enters an end state. This property can be expressed by the formula [](¬terminal).

2.4.2.1.2 Conditional Safety

An example of a conditional safety formula is `p->[]q`. In this case, whenever the state formula `p` becomes true, the state formula `q` must be true forevermore. Applied to an agent conversation, if a sender of a message receives a reply acknowledging receipt of said message, the sender's conversation will terminate and stay terminated.

2.4.2.2 Liveness Properties

Liveness properties have the form "good things will happen." Examples of liveness properties include termination or non-termination requirements in programs.

2.4.2.2.1 Guarantee Properties

Guarantee formulas state that some property will eventually happen. They guarantee that the event happens at least once, but make no promises of the event repeating. In fact, it doesn't matter if the event happens again, as long as it happens once. Therefore, guarantee formulas are used to ensure events happen at least once in the lifetime of a program execution, such as program termination.

An example of this property applied to an agent conversation is `<> end`. This means that a conversation eventually enters the `end` state and terminates. This concept is also used to check for the existence of livelock. If a process or conversation does not end when it is designed to end, it is evidently caught in a livelock situation.

2.4.2.3 Obligation Properties

Sometimes a safety or liveness property alone does not sufficiently describe the desired state of the system or conversation. In this case, a combination of the two types of properties is

needed. An *obligation* formula is a formula of the form [ ]p || <>q. As expected, this formula states that either p holds at all positions of a computation or q holds at some position.

2.4.2.3.1 Response Properties

An example of a response property is [ ]<>p. This property states that p can be satisfied infinitely many times in the computation, but at least once. Applied to an agent conversation, a response property would be used in the following scenario. A sender sends a message to a receiver and waits a specified amount of time for an acknowledgment. If the acknowledgment doesn't come, the message is sent again until an acknowledgment finally arrives.

2.4.2.3.2 Persistence Properties

Persistence properties are specified as <>[ ]p. This property states that all positions from a certain point on in a computation or conversation will satisfy p. Persistence formulas are used to describe the eventual stabilization of some state or property of the system or conversation. These properties allow an unspecified and varying delay until the stabilization occurs, but mandate that after occurring, it must be continuously maintained.

2.4.2.3.3 Reactivity Properties

Reactivity formulas are formed by a disjunction of a response formula and a persistence formula [ ]<>p || <>[ ]q. This formula states that either p occurs infinitely many times or q occurs all but a finite number of times.

2.4.3 Summary

Many properties of systems can be verified with an automated verification tool, including temporal properties. A system designer must first model the proposed system and then he can simply define a system's behavior over time using temporal formulas. Automated tools can then

search the entire state space of the system to verify that general communication faults are not present and the described temporal properties hold true.

2.5 Formal Languages and Automated Verification Tools

This section provides an overview of three formal languages and three automated tools that are used to model and verify communication systems. They are Communicating Sequential Processes (CSP), Failures-Divergence Refinement 2 (FDR2), Calculus of Communicating Systems (CCS), Concurrency WorkBench (CWB), Process Meta Language (Promela) and Spin. Together, these languages and tools help an analyst verify that a system design will perform correctly.

2.5.1 Communicating Sequential Processes

C.A.R. Hoare first described CSP in a 1978 paper. The basic ideas from his original paper were later adjusted and updated to produce a more flexible version of CSP (Hoare, 1985). As an example of CSP syntax, consider a clock that never does anything but tick. The keyword CLOCK describes the process and the keyword tick describes an event within the process (Hoare, 1985).

CLOCK = (tick->CLOCK)

This simple example illustrates a CSP recursive model. CSP allows the description of systems as a group of individual processes, which communicate with each other over channels. (Hoare eventually determined that component processes did not have to be sequential, but the name was already established.)

The modularization of CSP fits the structure of many problems very well. With CSP, an analyst models a system as a network of processes that communicate via messages along

unidirectional channels. The transfer of messages between processes is synchronous, which means the sending or receiving process stalls until the system transfers the message.

2.5.2 Failures-Divergence Refinement 2

FDR2 is a tool that allows an analyst to define a finite-state based system and then verify the system is correctly designed (Lowe, 1997). It is based on the theory of Communicating Sequential Processes (CSP). The theory of refinement in CSP enables a system engineer to describe a wide range of correctness conditions, including freedom from deadlock and livelock as well as safety and liveness properties.

Early versions of FDR could analyze systems with 10^7 states in a modest amount of time on standard workstations. The most current version of FDR2 incorporates hierarchical abstraction and compression routines that allow systems with very large state-spaces, ($7^{2^{\wedge 1024}}$) for example, to be analyzed in minutes.

FDR2 is simple to use and has extensive debugging facilities to support system development. If an error is detected during a verification process, FDR2 provides a description of the system state at the point where FDR2 detected the error, as well as the sequence of events that lead to the error.

2.5.3 Calculus of Communicating Systems

Robin Milner's work on CCS developed from an experiment in 1972. While working in the Artificial Intelligence Laboratory at Stanford, he tried to apply ideas learned while working on sequential programming to a concurrent programming language. However, he found this was not possible (Milner, 1989). One of the problems he ran into was because of an incorrect assumption. One way to decipher a sequential program is as a mathematical function over system memory states. If one knows the function corresponding to a particular program and the start

state, then one can figure out the end state. The only problem with this approach is that it assumes the program has exclusive control of the memory. If something interferes with the memory, then unpredictable states result.

Two programs that have the same function can behave very differently when subjected to the same interference. Milner gives a simple example where two lines of a computer program have exactly the same effect, in the absence of interference:

1) `x:=1`   2) `x:=0; x:=x+1`

However, suppose some other process at some unpredictable moment performs `x:=1`. Then the total effect of fragment 1 plus the other processes execution is different from that of fragment 2 plus the offending process. In fragment 2, the value of x could be either 1 or 2, depending on the given situation. This simple example demonstrates that in the presence of concurrency or interference, programs do not have exclusive rights to memory, but instead programs interact with each other while sharing memory.

This experiment prompted Milner to find an alternate theory in which communication was the focus. In 1977 Milner learned of Hoare's work with CSP and realized that he and Hoare both recognized that a new concept was needed, the concept of *indivisible interaction*. Milner also started a concept called *observation equivalence* of processes. The theory of observation equivalence was recorded in "A Calculus of Communicating Systems", published in 1980 (Milner, 1989).

An example of CCS follows.

```
C       =       in(x).C'(x)
C'(x)   =       out(x).C
```

Milner points out that agent names like C or C' can take parameters (Milner, 1989). In this case C' takes one parameter but C takes none. The prefix "`in(x).`" means a handshake takes place

where a value is received at port `in` and the variable `x` becomes equal to that value. `in(x).C'(x)` is an agent expression. It is required to perform the aforementioned handshake and then continue according to the definition of `C'`. The statement, `out(x).C`, is also an agent expression. This agent's behavior is to place the value of `x` at port `out` and then continue according to the definition of `C`.

2.5.4 The Concurrency WorkBench

The Edinburgh Concurrency Workbench is an automated tool designed to manipulate and analyze concurrent systems (Stevens, 1998). The CWB enables its users to check their systems in many different ways. The definition language for CWB is CCS. With CWB, users can perform tests on specified systems such as reachability analysis and model checking. Users can also verify systems defined with temporal properties. CWB allows users to interactively simulate the behavior of an agent. This is accomplished by guiding the agent through its state space in a controlled fashion.

2.5.5 Process Meta Language

Promela differs from the languages discussed thus far in that it is a modeling language. As such, it is used to abstractly model communication protocols (Holzmann, 1997). Promela is perfectly suited for modeling agent conversations. Conversations are modeled as processes, conversation paths are modeled as channels, and variables that may be used in a conversation can be defined and tested. All statements are either executable or blocked, waiting to execute. Statements may be blocked if the statement is a conditional statement and the condition is false. In this case, the statement blocks until the condition becomes true. This property provides a means of synchronizing communications between processes by causing one process (a responder)

to wait on a message sent by another process (the initiator) while in a specific state. The

initiating process may also block while waiting on a reply from the responding process.

Promela processes are defined with the word proctype. The following is an example of

a proctype declaration.

```
proctype ProcessA()
{
   byte newVariable;
   newVariable = 3
}
```

The name of this process is ProcessA and curly braces encapsulate the body of the

declaration. Promela declarations can contain zero or many statements as well as local variable

declarations. The proctype declaration above contains one local variable declaration,

newVariable, and a single statement: an assignment of the value 3 to the variable

newVariable. In Promela, semicolons and arrows '->' separate statements. Therefore, in the

above example no semicolon is needed after the last statement. The arrow is sometimes used as a

way to indicate a *causal* relation between two statements. For example:

```
byte newVariable = 2;
proctype ProcessA()
{
   (newVariable == 1) -> newVariable = 3
}
proctype ProcessB()
{
   newVariable = newVariable - 1
}
```

This example declares two processes, ProcessA and ProcessB. Since the variable

declaration newVariable is outside all processes, it is a global variable initialized to the value of

two. ProcessA contains two statements and ProcessB contains a single statement that

decrements newVariable by one. An assignment is always executable, so ProcessB does not

block and executes immediately. However, if a condition is not true, then the process is blocked

until the condition becomes true. Therefore, `ProcessA` is blocked at the condition (`newVariable==1`) until the value of `newVariable` is equal to one.

A `proctype` is only a process. It cannot run on its own. Something must start the process running. Spin uses a process called `init` to start other processes running. The `init` process is similar to a main procedure in Java programs. An `init` process declaration for the previous example would look as follows:

```
init
{
  run ProcessA();
  run ProcessB()
}
```

In this example, the keyword `run` kicks off the two processes. Parameters can also be passed when invoking processes with the `run` statement. For example, `run ProcessA(parameter1)`.

2.5.6 Spin

Spin is an automated verification tool from Bell Labs that operates on the Promela modeling language. It is designed to verify software instead of hardware and has been used to verify many distributed systems and communication protocols (Holzmann, 1997). Spin will detect deadlock, livelock, assertion violations, and many other communication centric errors.

Spin supports both synchronous and asynchronous communications by using channels to pass messages and varying the channel buffer size. If the channel buffer size is zero, then the communications are synchronous. If the channel buffer size is greater than zero, the communications are asynchronous.

With Spin, many types of simulations are possible. A user may choose to perform a random simulation, or a guided simulation. A user may also choose to perform a verification that

exhaustively searches the entire state space of the model for errors. If Spin finds an error, the user can then perform a guided simulation that will reproduce the condition that caused the error. This technique is very helpful for finding and pinpointing errors in models.

Spin can also catch correctness violations by checking for the existence of execution sequences that abort because an `assert` statement has been violated. An `assert` statement mandates that the asserted statement must remain true at all times.

To verify a system model is correct, Spin uses three specialized Promela states and analyzes temporal formulas like those mentioned in previous sections. End states, `progress` states, and `accept` states are used along with `never` claims to verify models. These features will be covered in the next few sections.

2.5.6.1 End states

If a process does not complete its processing before the system terminates, Spin flags the process as being in an invalid `end`-state. This is a common technique used to detect deadlock. If a system designer designs a process so that it can stop without completing, then the process has to be marked with an `end` label.

2.5.6.2 Progress states

Spin uses `progress` states to detect the presence of infinite overtaking by keeping track of how often Spin executes a process labeled with a `progress` label. Spin will produce an error if it cannot execute a `progress` labeled process an infinite number of times. In other words, any process labeled with a `progress` label cannot remain blocked indefinitely from executing.

An example of the use of a `progress` label is:

```
proctype ProcessA(){
  do
  :: chanAtoB!p -> progress: chanAtoB?v
  od}
```

The presence of the `progress` label requires the statement `chanAtoB?v` be executed infinitely many times. The only way this statement can be executed infinitely many times is if the statement `chanAtoB!p` can also be executed infinitely many times.

2.5.6.3 Accept states

The `accept` state is treated exactly the opposite as the `progress` state and is used to detect correctness of temporal property specifications. If Spin enters an `accept` state an infinite number of times, an error is produced. A user can label a "trap" state with an `accept` label and then Spin will check an infinite number of times if it can enter the trap state. If it can, then the condition leading to the trap state has been met and Spin has succeeded in catching the error condition.

The following process demonstrates the use of an `accept` state. This process should eventually block at the beginning of the statement `chanAtoB!p`. If this process did *not* eventually block at this statement, then the process would run forever, and this would cause Spin to produce an error. The `accept` state would be visited infinitely often because the process would run forever, thus creating the error condition.

```
proctype ProcessA()
{ do
  :: chanAtoB!p -> accept: chanAtoB?v
  od
}
```

2.5.6.4 Never claims

Spin uses `never` claims to define temporal formulas. These `never` claims are then used to check for undesirable or illegal state properties. Spin will produce an error if it finds any execution sequence that ends where the `never` claim has terminated by reaching the closing brace of its body. As detailed in an earlier section, Spin will produce an error if there is an

23

execution sequence that visits infinitely often an `accept` state. Combining `never` claims and `accept` states, Spin can detect illegal infinite (cyclic) behavior by labeling a block of statements in a `never` claim with an `accept` label, creating an `accept` state, and then checking the selected statements an infinite number of times to see if Spin can enter the blocked `accept` state.

2.5.6.5 Example claim

If p and q are two boolean variables and the temporal claim is made that *"along every computation, each system state in which p is true is eventually followed by the case when q is true,"* then the following `never` claim verifies whether there are any violations of the temporal claim.

```
never
{
  do
    :: p -> break
    :: skip
  od;
  accept:
  do
    :: !q
  od
}
```

The first `do` loop terminates only when the variable p becomes true. According to our claim, the variable q should eventually become true. The second `do` loop (hence the `never` claim) will never terminate and cannot be broken out of. The `never` claim continuously checks the system state to see if q has become true. The `never` claim either eventually blocks because the variable q becomes true (which is the desired behavior) and the `accept` state cannot be entered, or Spin continuously enters the `accept` state because q is not true. If Spin can enter the `accept` state without ever being blocked, this is an error. Because Spin guarantees an exhaustive

search of the system state space, if there is any violation of our claim, Spin will detect it. *If the never claim ends in the* accept *state, then an error has been detected.*

2.5.7 Summary

Each of the formal languages covered can accurately portray a system and describe the system's behavior. The difference in the languages is the ease of use and understandability of the language, as well as the automated tools that support the language. Promela is a modeling language, and thus resembles a programming language rather than a formal language. This feature makes Promela easier to understand for most computer scientists.

All three of the automated verification tools covered here can verify a system is deadlock free. They can also verify safety properties and liveliness properties. The basic difference in the systems lies in their input language. Therefore, the choice of which system to use depends primarily on the choice of design language.

III. Methodology

3.1 Introduction

Currently, the only way to formally verify the agent conversations designed in a multi-agent environment with an automated tool is for someone to translate, by hand, the design into a formal language and then run the verifier on this formal representation. *Most people believe formal methods are too difficult to understand and use in this manner* (Hinchey, 1999). The challenge then is to automatically generate the formal representation of a conversation from the design in the multi-agent development environment. Then, using an automated tool, verify this representation is free from undesirable communication properties such as deadlock.

As stated in Section 1.3, the goal of this research is to develop a formal methodology and technique to verify that the communication protocols defined in a multi-agent environment are valid. This chapter outlines steps that can be used to apply this research to any multi-agent development environment. Section 3.2 explains how an agent conversation is modeled with a state transition diagram. Section 3.3 explains how the state transition diagram can then be converted into a set of state tables. The task of creating a formal representation of the state transition diagram from the state table is described in Section 3.4. The process of verifying the formal representation with an automated tool is detailed in Section 3.5. Figure 3 is a top-level view of the overall process.

3.2 Modeling Agent Conversations with State Transition Diagrams

According to Roger Pressman,

> The state transition diagram indicates how the system behaves as a consequence of external events. To accomplish this, the state transition diagram represents the various modes of behavior (called states) of the

26

system and the manner in which transitions are made from state to state. (Pressman, 1997)

```
┌─────────────────────────────┐
│      Model Conversation     │
│            With A           │
│    State Transition Diagram │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│    Convert State Transition │
│         Diagram To A        │
│          State Table        │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        Create Formal        │
│     Representation From      │
│          State Table        │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│        Verify Formal        │
│    Representation With An    │
│        Automated Tool       │
└─────────────────────────────┘
```
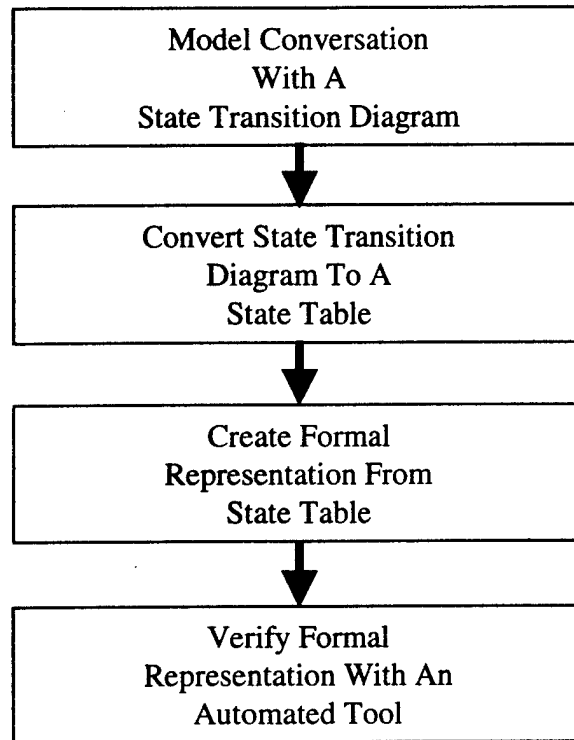
Figure 3: Top Level View of Methodology

An agent conversation consists of an *initiator* side and a *responder* side. Both sides of the conversation move through various states by sending and receiving messages. Eventually, both sides of the conversation should end up in their respective "end" states and the conversation will be completed. It is the state transition diagram that allows us to visualize the various states a conversation goes through and it records the events that cause the conversation to move from state to state.

As shown in Chapter 2, Figure 4 illustrates one side of a conversation and Figure 5 illustrates the complimentary side of the conversation. The two sides make up one complete

27

conversation, which may be part of a much larger system (or set) of conversations. The conversation is shown here again for easy reference.

The responder side of the *SendInfo* conversation has four states and the transitions complement the transitions in the initiator side of the conversation. The next step in the modeling process is to convert the above state transition diagrams into a state table.
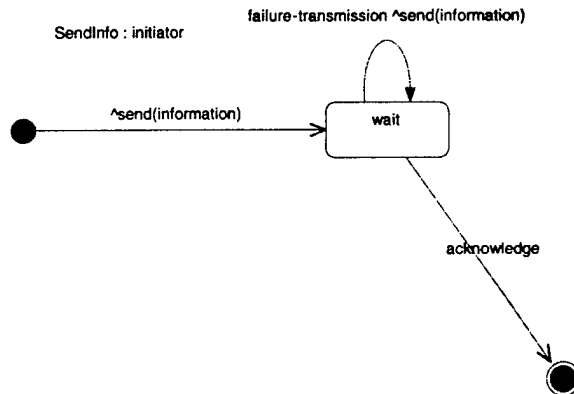
Figure 4: Initiator Half of Conversation SendInfo
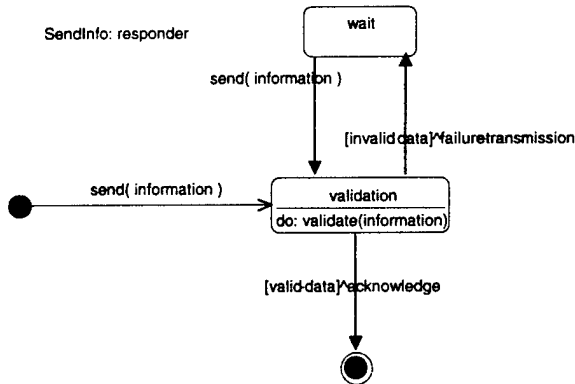
Figure 5: Responder Half of Conversation SendInfo

3.3 Converting a State Transition Diagram to a State Table

A state table is a textual representation of a graphical state transition diagram. The advantage a state table has over a state transition diagram is that it can be parsed easily. This feature is critical when Promela source code has to be generated.

The state table is built from the transition labels on the transition arrows of a state transition diagram. The state table is simply an ordering of all the transitions possible in a state transition diagram. The state table is ordered so all transitions pertaining to a particular state are together. The state table also must begin with the *Start* state and end with the *End* state. Normal state tables do not have these requirements, but they are necessary here for automatically building Promela source code. The format of the state table should mirror that of the transition labels in a state transition diagram. However, each entry in the state table needs to know the state the transition is coming from and the state it is going to, even if it is the same state. One solution to this problem is to add to the beginning of the state table entry the current state of the transition while adding to the end of the entry the next state the transition will enter. The different fields of the state table entry should be separated by a semicolon or some other character for ease in parsing the table later. Figure 6 illustrates a state table using the *SendInfo* conversation in Figures 4 and 5. In this state table, a name is given to both halves of the conversation and this name inserted at the beginning of each line. This naming convention will be used to create Promela code later on.

```
SendInfoResponder;startState;send;null;null;validationState
SendInfoResponder;validationState;null;invalidData;
    failureTransmission;waitState
SendInfoResponder;validationState;null;validData;acknowledge;
    endState
SendInfoResponder;waitState;send;null;null;validationState
SendInfoResponder;endState;null;null;null;null
SendInfoInitiator;startState;null;null;send;waitState
SendInfoInitiator;waitState;failureTransmission;null;send;waitState
SendInfoInitiator;waitState;acknowledge;null;null;endState
SendInfoInitiator;endState;null;null;null;null
```

Figure 6: State Table of Conversation SendInfo

Each line of the state table contains the following information: *process name* (consisting of the *conversation name* and the *participant's name*), *current state, received message, guard*

condition, transmitted message, and *next state*. Each entry in the state table must be unique to prevent duplication of Promela code.

The state table provides a textual representation of the state transition diagram. The state table is now used to build a formal representation of the state transition diagram by converting the state table into Promela source code. The following section demonstrates how Promela is used to model an agent's conversation.

3.4 Creating Promela Code from a State Table

Modeling a conversation with Promela is not as difficult as one would think. However, creating the Promela code using as input a state table requires a method of parsing the state table and automatically creating the source code. In this section, the *SendInfo* conversation is modeled. Each Promela statement will be described as it is used.

3.4.1 Message Type Declarations

The first line of Promela code needed is the message type declarations. Promela has a type called `mtype` that allows a programmer to declare constants without assigning values to the constants. The declaration looks like this:

```
mtype={failureTransmission,send,invalidData,validData,acknowledge};
```

Promela does not allow hyphens in declarations, thus the word `failureTransmission` instead of `failure-transmission`. These values are found by searching through the state table and creating a vector of messages by examining the *received message, guard condition*, and *transmit message* fields.

3.4.2 Channel Declarations

The next declaration required is the channel the messages will use. Promela allows for synchronous or asynchronous transmissions. The channel declaration looks like this.

```
chan bus1 = [1] of {mtype};
```

This declaration states that a variable bus1 is of the type chan, and it can hold one message in its buffer. Only messages of type mtype can be sent on this channel. If the [1] was replaced with [0], then no messages could be buffered and all messages would have to be taken off the channel (received) before another message could be placed on the channel (transmitted). The channel declarations are determined by the number of conversations in the state table. If only one conversation is in the state table, then only one channel declaration must be made. However, if for instance three conversations are contained in the state table, then three channels must be used to prevent messages from interfering with each other.

3.4.3 Process Declarations (Proctypes)

The next step is to create processes that emulate each side of the conversation. Promela has a construct called a proctype that models each half of a conversation. Each process contains all of the states for one side of the conversation. The processes are designed to begin in the startState and end in the endState, while moving from states only if explicitly directed to do so. Figure 7 shows the proctype declaration for the *responder* side of the *SendInfo* conversation, while Figure 8 shows the *initiator* side of the same conversation.

The keyword proctype declares a procedure. The state labels all end with a colon. The do..od loops trap the flow of control inside their respective states. Two ways to exit a do..od loop is with a goto statement or a break statement. The goto transfers control to another state while the break just exits the loop and falls through into the next state. For obvious reasons, it is unacceptable to fall into another state unless explicitly directed to do so. An exclamation point (!) after the channel variable bus1 signifies the message send has been placed on the channel. The arrow (->) is a statement separator and serves as an implication symbol. If the statement

before the arrow is executed then the statement after the arrow is also executed. The semicolon (;) is also a statement separator but carries no implications. Finally, a question mark (?) after the channel variable bus1 signifies the message following the question mark is taken off of the channel via a receive action.

```
proctype SendInfoResponder()
{
  progressStartState:
    do
    :: bus1?send -> goto progressvalidationState
    od;
  progressvalidationState:
    do
    :: invalidData->bus!failureTransmission;goto
       progresswaitState
    :: validData -> bus1!acknowledge; goto progressendState
    od;
  progresswaitState:
    do
    :: bus1?send -> goto progressvalidationState
    od;
  progressEndState:
    do
    :: break
    od;
}
```

Figure 7: Process SendInfoResponder

```
proctype SendInfoInitiator()
{
  progressStartState:
    do
    :: bus1!send -> goto progresswaitState
    od;
  progresswaitState:
    do
    :: bus1?failureTransmission-> bus!send; goto
       progresswaitState
    :: bus1?acknowledge -> goto progressendState
    od;
  progressEndState:
    do
    :: break
    od;
}
```

Figure 8: Process SendInfoInitiator

32

3.4.4 Process Declarations (Init)

Now that the processes representing the two halves of the conversation have been modeled, a process needs to be created that will start the conversation processes running. This process is called an init process. Figure 9 shows what the init process looks like for the *SendInfo* conversation.

```
init
{   atomic
    {
        run SendInfoResponder();
        run SendInfoInitiator()
    }
}
```

Figure 9: Init Process for SendInfo Conversation

The keyword atomic mandates all statements enclosed within its brackets will be executed without interruption by external processes. The keyword run starts the processes running and these processes are run in parallel. Figure 10 shows the complete Promela code for the *SendInfo* conversation.

3.4.5 Verifying Message Sequences

Sequence diagrams (Rational, 1997) are beneficial for real-time specifications and for complex scenarios. They show the explicit sequence of messages between agents and can exist in a generic form (all the possible sequences of messages) or an instance form (one actual sequence consistent with the generic form). Sequence diagrams show the big picture in the grand scheme of agent conversations.

Listing desired messages between conversations in a specified order creates a message sequence. Sequence diagrams represent interactions among agents within a system to achieve a desired operation or result. A graphical representation of a message sequence is called a message sequence chart (Rational, 1997). Figure 11 shows a valid message sequence chart encompassing

two conversations (SendInfo and CollectData) between three agents (Commander, Mission Cntrl,

and Data Collection). Not all of the messages that could be sent in these conversations need be

included in the message sequence chart.

```
mtype = {failureTransmission, send, invalidData, validData,
         acknowledge};
chan bus1 = [1] of {mtype};
proctype SendInfoResponder()
{
  progressStartState:
    do
    :: bus1?send -> goto progressvalidationState
    od;
  progressvalidationState:
    do
    :: invalidData -> bus!failureTransmission; goto
       progresswaitState
    :: validData -> bus1!acknowledge; goto progressEndState
    od;
  progresswaitState:
    do
    :: bus1?send -> goto progressvalidationState
    od;
  progressEndState:
    do
    :: break
    od;
}
proctype SendInfoInitiator()
{
  progressStartState:
    do
    :: bus1!send -> goto progresswaitState
    od;
  progresswaitState:
    do
    :: bus1?failureTransmission -> bus!send; goto
       progresswaitState
    :: bus1?acknowledge -> goto progressendState
    od;
  progressEndState:
    do
    :: break
    od;}
init
{ atomic
  { run SendInfoResponder();run SendInfoInitiator() }}
```

Figure 10: Complete Promela Code for SendInfo Conversation
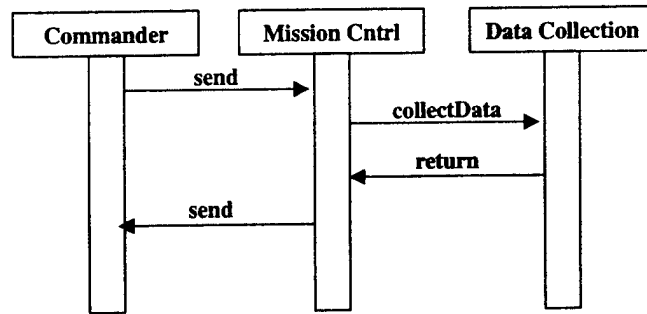
Figure 11: Message Sequence Chart

Message sequences are converted to a table similar to a state table as shown in Figure 12.

The format of the message sequence table is *Conversation Name*; *Conversation From Participant*; *Conversation To Participant*; *Message*. When checking for a message sequence the sequence is defined in a Promela never claim and checked for its existence. A never claim is a special type of process that is optional and, if it exists, is used to detect undesirable behavior. If a message sequence defined in a never claim is found, Spin will generate an error. Of course, this is not really an error because we want to verify the sequence exists and the error condition has confirmed the sequence does indeed exist. Figure 13 is the never claim for the message sequence table of Figure 12.

```
SendInfo;Responder;Initiator;send
CollectData;Initiator;Responder;collectData
CollectData;Responder;Initiator;return
SendInfo;Initiator;Responder;send
```

Figure 12: Message Sequence Table

A key difference in the modeling of a message sequence and a conversation is the way message events are detected. In a conversation, the channel that messages are transmitted on is constantly monitored and messages must be placed on the channel and taken off the channel in a predetermined order. In a message sequence, the channel is monitored but only desired messages are detected.

Many messages may be placed on the channel and taken off the channel before a desired message is detected as part of a particular sequence. Modeling sequences in this fashion provides great flexibility in detecting message sequences that span multiple conversations.

```
never
{
    State0:
        do
        :: SendInfo?[send] -> goto State1
        :: skip
        od;
    State1:
        do
        :: CollectData?[collectData] -> goto State2
        :: skip
        od;
    State2:
        do
        :: CollectData?[return] -> goto State3
        :: skip
        od;
    State3:
        do
        :: SendRawIntel?[send] -> goto State4
        :: skip
        od;
    State4:
        do
        :: SendInfo?[send] -> goto accept
        :: skip
        od;
    accept:
        skip
}
```

Figure 13: Never Claim for Message Sequence Verification

The completed Promela source code is now saved and will be used as input for the verification tool Spin. The verification of the *SendInfo* conversation is covered in the next section.

3.5 Verifying a Communication Protocol Using Spin

There are three steps in running Spin: 1) Compile the source code, 2) Generate the analyzer files, and 3) Execute the analyzer.

3.5.1 Compile the Source Code

Spin is invoked by passing it the file name of our Promela code. This command looks as follows:

```
redir -o error spin -a verify
```

The `redir -o error` portion of the above command uses a utility provided by the C compiler that will redirect the output of the spin command to a file called error. The `-a` parameter generates a protocol specific analyzer. Spin's output is a set of C files, named pan (protocol analyzer).

3.5.2 Generate the Analyzer Files

The second step in running Spin is to compile the pan files with a C compiler to produce the analyzer (`pan.exe`), which is then executed to perform an analysis of the protocol. The command required to compile the pan files is as follows:

```
gcc -DEBITSTATE -DSAFETY -o pan pan.c
```

The `-o` parameter guarantees an exhaustive state space search for errors. The `-DEBITSTATE` parameter uses a memory efficient bit state space method to prevent exhausting the memory available on some machines. The `-DSAFETY` parameter decreases the overhead associated with liveness properties when only checking for safety properties. In this case, the check is for deadlock, which is a safety property.

If checking for non-progress states, a different command must be used. It is not possible for Spin to check for both deadlocks and non-progress states at the same time. The command needed is as follows:

```
gcc -DNP -DBITSTATE -o pan2 pan.c
```

In this command, the -DNP parameter directs Spin to check for non-progress cycles instead of deadlocks.

There is one more command that can be used to analyze a conversation. If a never claim is used in the model, then the -DSAFETY parameter cannot be invoked. This is because a never claim can incorporate more than just safety properties. It is possible to check for a *message sequence* with Promela/Spin using a never claim. Figure 13 is a message sequence trace that contains the message sequence of Figure 11. The command that must be used when a conversation is modeled this way is as follows:

```
gcc -DEBITSTATE -o pan pan.c
```

Notice that the command is just like the command to check for deadlocks, but without the -DSAFETY parameter.

3.5.3 Execute the Analyzer

The third step in running Spin is to execute the analyzer. The pan files are compiled into an executable file called pan.exe. The pan.exe file is the analyzer that when executed analyzes the compiled protocol. The command to execute the analyzer is as follows:

```
redir -o output.txt pan.exe
```

This is the command to use when checking for deadlocks. When running the pan.exe file, a trace file (verify.trail) is created if an error is found in the protocol. This trace file

can then be examined by Spin to pinpoint the location of the error. The command to generate a

sequence trace based on the trail file is as follows:

```
redir -o trace.txt spin -t -c verify
```

```
proc 0 = :init:
proc 1 = SendInfoResponder
proc 2 = SendInfoInitiator
proc 3 = CollectDataInitiator
proc 4 = CollectDataResponder
proc 5 = CollectDataInitiator
proc 6 = CollectDataResponder
proc 7 = SendRawIntelResponder
proc 8 = SendRawIntelInitiator
proc 9 = SendInfoResponder
proc 10 = SendInfoInitiator
q\p  0  1  2  3  4  5  6  7  8  9  10
  1  .  .  .  .  .  .  .  .  .     SendInfo!send
  1  .  .  .  .  .  .  .  .  .  SendInfo?send
  1  .  .  .  .  .  .  .  .  .  SendInfo!failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo?failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo!send
  2  .  .  .  .  .  .  .  .  SendRawIntel!send
  1  .  .  .  .  .  .  .  .  .  SendInfo?send
  1  .  .  .  .  .  .  .  .  .  SendInfo!failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo?failureTransmission
  2  .  .  .  .  .  .  .  SendRawIntel?send
  1  .  .  .  .  .  .  .  .  .     SendInfo!send
  1  .  .  .  .  .  .  .  .  .   SendInfo?send
  1  .  .  .  .  .  .  .  .  .  SendInfo!failureTransmission
  2  .  .  .  .  .  .  .  SendRawIntel!failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo?failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo!sen
  1  .  .  .  .  .  .  .  .  .  SendInfo?send
  2  .  .  .  .  .  .  .  SendRawIntel?failureTransmission
  1  .  .  .  .  .  .  .  .  .  SendInfo!failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo?failureTransmission
  1  .  .  .  .  .  .  .  .  .     SendInfo!send
  1  .  .  .  .  .  .  .  .  .  SendInfo?send
  1  .  .  .  .  .  .  .  .  .  SendInfo!acknowledge
  1  .  .  .  .  .  .  .  .  .     SendInfo?acknowledge
  2  .  .  .  .  .  .  .  .  SendRawIntel!send
  2  .  .  .  .  .  .  .  SendRawIntel?send
  2  .  .  .  .  .  .  .  SendRawIntel!failureTransmission
  3  .  .  .  .  .  CollectData!collectData
  2  .  .  .  .  .  .  .     SendRawIntel?failureTransmission
  2  .  .  .  .  .  .  .  .  SendRawIntel!send
  2  .  .  .  .  .  .  .  SendRawIntel?send
  2  .  .  .  .  .  .  .  SendRawIntel!failureTransmission
  3  .  .  .  .  .  .  CollectData?collectData
  2  .  .  .  .  .  .  .  .  SendRawIntel?failureTransmission
  2  .  .  .  .  .  .  .  .  SendRawIntel!send
  2  .  .  .  .  .  .  .  .  SendRawIntel?send
  3  .  .  .  .  .  .  CollectData!collectionFailure
  2  .  .  .  .  .  .  .  SendRawIntel!failureTransmission
  2  .  .  .  .  .  .  .     SendRawIntel?failureTransmission
  2  .  .  .  .  .  .  .     SendRawIntel!send
  2  .  .  .  .  .  .  .  SendRawIntel?send
  3  .  .  .  .  .  CollectData?collectionFailure
  2  .  .  .  .  .  .  .  SendRawIntel!failureTransmission
  2  .  .  .  .  .  .  .     SendRawIntel?failureTransmission
  2  .  .  .  .  .  .  .  .  SendRawIntel!send
```

Figure 13: Message Trace of Message Sequence Verification

The -t parameter directs Spin to follow the simulation trail in the tail file (verify.trail). The -c parameter tells Spin to put the simulation output in columnated order. If checking for non-progress errors, the command to execute the analyzer is as follows:

```
redir -o newpan2.txt pan2.exe -l
```

In this command, the -l parameter tells the analyzer to find non-progress cycles. If checking for a message sequence with a never claim that contains an accept state, then a different command must be used. The command is as follows:

```
redir -o newpan.txt pan.exe -a
```

In this command, the -a parameter tells Spin to find acceptance cycles, which would have been declared inside the never claim. Note that a never claim can be declared without an acceptance state. However, Spin appears to find an error faster if an acceptance state is used.

3.6 Interpreting Results

The only thing left to do is display the output.txt file for any error messages, and if there were any errors, the trace.txt file for the detailed trace. The output will list any errors as well as the quantity of errors. Figure 14 shows an example of the Spin output with no errors.

The messages generated by Spin show that a full statespace search was performed for assertion violations and invalid end states. The search reached a depth of 12 levels and found no errors. This conversation model contained three processes, and none of them had states that were unreachable during the simulations.

If errors were detected during the verification process, text files are created that contain the detailed error information. If a deadlock condition occurs, Spin generates an invalid end-state error for each state that is deadlocked. If a state is never entered into, then Spin generates a non-

progress state error message. Finally, if a message sequence is not detected and an error generated, then the message sequence does not exist and a true error is found. Figure 15 shows the output generated by Spin when a deadlock condition is inserted into the *SendInfo* conversation by changing one of the transmitted messages to a received message.

```
(Spin Version 3.2.4 -- 10 January 1999)
        + Partial Order Reduction
Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance    cycles    - (not selected)
        invalid endstates +
State-vector 28 byte, depth reached 12, errors: 0
        12 states, stored
         1 states, matched
        13 transitions (= stored+matched)
         1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
1.493         memory usage (Mbyte)
unreached in proctype SendInfoResponder
        (0 of 24 states)
unreached in proctype SendInfoInitiator
        (0 of 18 states)
unreached in proctype :init:
        (0 of 4 states)
```
Figure 14: Spin Output of SendInfo Conversation

In this output, only one error is detected. However, it was an invalid end-state caused by a deadlocked state in the conversation. Spin generated a file called verify.trail that can be used to recreate the message trace that caused the deadlock condition. This is very useful in troubleshooting the condition that caused the error.

Figure 16 is the output generated by Spin when checking for non-progress states. Non-progress states are detected if any state labeled with the keyword progress is not entered into. This output did not detect any errors, but did note that two states in the procedure SendInfoResponder and two states in SendInfoInitiator were not reached. This error

was caused because the conversation was deadlocked, and thus the conversation could not proceed to these states and complete the conversation.

```
pan: invalid endstate (at depth 5)
pan: wrote verify.trail
(Spin Version 3.2.4 -- 10 January 1999)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never-claim            - (none specified)
    assertion violations +
    cycle checks           - (disabled by -DSAFETY)
    invalid endstates +

State-vector 24 byte, depth reached 8, errors: 1
        9 states, stored
        1 states, matched
       10 transitions (= stored+matched)
        1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493    memory usage (Mbyte)
```

Figure 15: Spin Output of Detected Deadlock

3.7 Summary

This chapter described the methodology used to verify agent conversations in a multi-agent system. The process began by modeling the conversations using a state transition diagram. The state transition diagram was then converted into a state table where it was parsed into the Promela modeling language. Finally, Spin was run against the Promela code and deadlock and non-progress errors where checked for. Also demonstrated was how Promela and Spin could be used to verify message sequences by declaring a never claim and checking for the existence of the desired message sequence.

```
(Spin Version 3.2.4 -- 10 January 1999)
  + Partial Order Reduction

Bit statespace search for:
   never-claim          +
   assertion violations + (if within scope of claim)
   non-progress cycles  + (fairness disabled)
   invalid endstates - (disabled by never-claim)

State-vector 32 byte, depth reached 16, errors: 0
       10 states, stored
        2 states, matched
       12 transitions (= stored+matched)
        2 atomic steps
hash factor: 381300 (expected coverage: >= 99.9% on avg.)
(max size 2^22 states)

3.066    memory usage (Mbyte)

unreached in proctype SendInfoResponder
   line 21, state 21, "goto"
   line 24, state 24, "-end-"
   (2 of 24 states)
unreached in proctype SendInfoInitiator
   line 38, state 15, "goto"
   line 41, state 18, "-end-"
   (2 of 18 states)
unreached in proctype :init:
   (0 of 4 states)
```

Figure 16: Spin Output of Detected Non-progress State

IV. Implementation

4.1 Introduction

Chapter 3 described how this research could be applied to a generic multi-agent development environment. This chapter outlines the steps taken to implement the generic methodology in AFIT's agentTool multi-agent development environment. Section 4.2 provides an overview of how multiple conversations are verified using agentTool. Section 4.3 steps through three examples of how to verify multiple conversations. The first example will not contain any errors. The second example will contain a deadlock condition and will contain non-progress states (states that are not entered into). The third example will demonstrate how a message sequence is verified. Finally, Section 4.4 gives an analysis of the types of errors that are detected and reported by agentTool, and perhaps more importantly, those errors that are not detected and reported.

4.2 Verification Overview

In agentTool, a conversation takes place between two agents. Therefore, the first step in verifying a conversation is to create two agents and establish a conversation between them. After the conversation is established, the two sides of the conversation must be defined. The agentTool environment automatically creates a start state and an end state for each side of the conversation. All the conversation designer must do is fill in the required states and transitions for each side of the conversation.

The conversation definition process is repeated until all necessary conversations are completed. The verification process is invoked by clicking on the Command pull down menu and

choosing the Verify Conversations option. Figure 17 is a Data Flow Diagram (DFD) of the entire verification process.
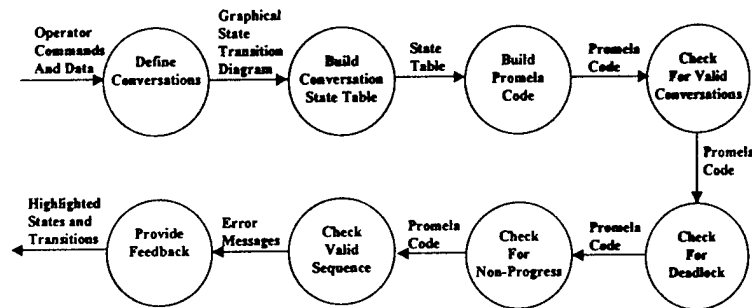
Figure 17: Verification Data Flow Diagram

The next section provides details of each process listed in Figure 17. After the conversations have been verified, feedback is provided to the user by means of a text window that contains useful and meaningful messages while highlighting states and transitions on the state transition diagrams where errors have been detected. As often happens with source code and compilers, a single error may generate many error indicators. For this reason, many states and transitions may be highlighted when only one or two is actually in error.

4.3 System Design

The conversation verification subsystem of agentTool was implemented using Java, text files, and batch commands. Each step of Figure 17 is detailed below to demonstrate how the step was actually implemented.

4.3.1 Define Conversations

Conversations are designed in agentTool using state transition diagrams. The diagrams are built using graphical tools. Conversation states and transitions between states have properties associated with them that are defined by the system designer. This part of the agentTool research effort is documented in Wood's Thesis (Wood, 2000).

4.3.2 Build Conversation State Table

The state transition diagram must be converted into a state table before automatically generating Promela source code. Wood's thesis addresses how the values for each state transition are derived and a state table created. Each entry in the state table contains the *conversation name*, the *participant*, the *current state*, the *received message* (if it exists), the *guard condition* (if it exists), the *transmitted message* (if it exists), and the *next state*. Every transition in the state transition diagram is mapped to an entry in the state table. The state table is actually a vector of transitions that can be analyzed to build the Promela source code.

It is important the state table be ordered on conversation states so that all of a conversation's information is contiguous (sequential without interruption). Therefore, once a state table has been created it is sorted so all of a given state's transitions are together in the table.

4.3.3 Build Promela Code

Chapter 3 explains the general process of automatically building Promela source code from a state table. The process takes five steps: 1) declare `mtype` variables, 2) declare `channels`, 3) build `proctypes`, 4) build `init` procedure, and 5) build `never` claim.

The Promela source code is saved in a text file. Before declarations can be made, the text file must be created and opened. The name of the text file is simply `Goverify`.

4.3.3.1 Declare mtype Variables

Received messages, guard conditions, and transmitted messages all must be declared as `mtype` variables. To find these variables, the state table vector is searched one transition at a time and appropriate variable names added to a new `mtype` vector. Every received message, guard condition, and transmitted message is compared to the variables in the `mtype` vector. If the variable already exists in the vector, it is passed over. However, if it does not exist in the vector,

it is added. After one compete pass through the state table, the `mtype` vector contains a list of the `mtype` variables with no duplicates.

The `mtype` declarations are the first entries in the Promela source code. The text string `mtype = {` is printed along with the contents of the `mtype` vector delimited by commas. After the `mtype` vector has been printed, the declaration is completed by printing `};` and starting a new line of the source code.

4.3.3.2 Declare Channels

Channels are the communication lines between two halves of a conversation. Therefore, a channel exists for every conversation in the state table. The channel declarations are made by first printing the text string `chan`. Then the state table is searched sequentially and every conversation name printed, comma delimited. The declaration is completed by printing `= [1] of {mtype};` and starting a new line of source code.

4.3.3.3 Build Proctypes

A `proctype` declaration must be made for each side of a conversation. The state table is ordered so that all the transitions for a conversation's participant are together. The first transitions are those from the start state and the last entry for each participant in the state table is the end state transition.

The state table vector is read sequentially and only one pass through the vector is required to create all the `proctypes`. For each `proctype` declaration, the text `proctype` is printed followed by the conversation name concatenated with the participant's name. This technique creates a unique `proctype` name for each side of every conversation. The initial line of the declaration is finished with the text `()` and an opening brace printed on a new line. Each state in the `proctype` is declared by printing the text `progress` followed by the state name and

47

finished with a colon. The next line of the state declaration contains the text do, which begins a do loop. Every line of text within the do loop contains the text :: followed by the proper formatting of the transition. After all the transitions for a given state are printed, the do loop is terminated by printing the text od;. After all the states have been printed, the proctype declaration is completed by printing a closing brace. This process repeats until all proctypes have been generated.

4.3.3.4 Build init Procedure

The init procedure is declared by printing the text init followed by an opening brace on a new line. The key word atomic is then printed followed by another opening brace on a new line. The state table is then read sequentially and a line printed for each conversation half (two entries per conversation). Each line contains the keyword run followed by the conversation name concatenated with the participant name and ended with parentheses. Each run statement must be separated by a semicolon. After all the run statements are written, two closing braces, each on its own line, must be printed.

The init procedure is the last part of the Promela source code that is created unless checking for a valid message sequence. Then, in addition to the above procedures, a never claim must be declared.

4.3.3.5 Build Never Claim

A never claim follows the init procedure. It is declared by first printing the keyword never followed by an opening brace on a new line. The never claim is built by reading a message sequence table. There must be a state in the never claim for each entry in the message sequence table. The states are declared by first labeling the state with the text State and appending to it an integer beginning with 0 and incrementing the integer by 1 for every new state

created. Each state label must end with a colon. Each state is made up of a do loop that contains two entries.

The first entry is the channel name the message is expected to traverse appended with a question mark to signify receiving a message on that channel. Appended to the channel name and question mark is the message name enclosed in brackets. This method of detecting a message on a channel allows unwanted messages to pass until the desired message is detected. An arrow is appended to the bracketed message and a goto statement that directs the conversation to the following state if the correct message is detected.

The second entry is a skip statement that keeps the never claim in the current state until the desired message is detected. The state declaration is finished by ending the do loop with the text od;.

After all the states have been printed, an accept state must be declared. The accept state traps the never claim until all the conversations have terminated. The accept state is created by printing the keyword accept: followed by the keyword skip on a separate line. The never claim is then completed by printing a closing brace. The completed Promela source code is now saved in the text file verify for use with Spin.

4.3.4 Check for Valid Conversations

When Spin is started, the Promela source code is first checked for syntactical errors. Syntactical errors such as invalid characters in variable names will cause Spin to generate an error file that contains the error messages. If after running Spin the error file contains messages, they are displayed in the message window for the user to analyze. The command to run Spin against the Promela source code created above is:

```
Spin.exe -a verify
```

The -a parameter tells Spin to create an analyzer specific to the protocols specified in the file `verify`. Syntactical errors must be corrected before the conversations can be verified. If no error messages are reported, Spin creates the appropriate files that can be used to generate an executable analyzer.

4.3.5 Check for Deadlock

Once the analyzer files have been created, an executable analyzer file must be created. This is accomplished by compiling one of the newly generated files (`pan.c`) into an executable file (`pan.exe`). The command required is:

```
gcc.exe -DEBITSTATE -DSAFETY -o pan pan.c
```

The `gcc` command invokes a standard C compiler. The -DEBITSTATE parameter uses a memory efficient bit state space method to prevent exhausting the memory available on some machines. The -DSAFETY parameter decreases the overhead associated with liveness properties when only checking for safety properties. The -o parameter guarantees an exhaustive state space search for errors.

Now `pan.exe` can be executed and the protocol specific files analyzed. The command required for this is simply:

```
pan.exe
```

Spin displays the results of the analysis by default to the computer screen. However, the output can be redirected to a text file by using a C utility, `redir`. The command to accomplish this task is:

```
redir -o output.txt pan.exe
```

The -o parameter is used by the `redir` command and states the output should be directed to the file `output.txt`.

4.3.6 Check for Non-Progress

The check for non-progress is similar to that for deadlock. However, special parameters must be used because Spin cannot check for both deadlock and non-progress with the same command. The command required is:

```
gcc.exe -DNP -DEBITSTATE -o pan pan.c
```

In this command, the -DNP parameter directs Spin to check for non-progress cycles instead of deadlocks. The newly created pan.exe must be executed to actually perform the analysis, but the procedure is the same as in Section 4.3.5.

4.3.7 Check Valid Sequence

If checking for a valid message sequence, a slight modification to the deadlock check is required. Since a never claim is declared when checking for valid message sequences, the check cannot only be for safety properties. The check must now also include checking for liveness properties associated with the never claim. The command required is:

```
gcc -DEBITSTATE -o pan pan.c
```

The command is exactly like the check for deadlocks except the -DSAFETY parameter is missing and cannot be used. The newly created pan.exe file must be executed to analyze the protocol specific files and generate the appropriate output.

4.3.8 Provide Feedback

Feedback is provided to the system designer through a text based message window and through graphical highlighting of the state transition diagram. When executing the pan.exe file, the output is redirected to a text file. The contents of the text file are then copied into the message window enabling the system designer to see the results of the analysis. Sometimes the Spin output is difficult to interpret for novice users, so the output is automatically parsed and

states and transitions are highlighted to assist the user in locating errors. During the verification process, a vector containing all known deadlock transitions and non-progress states is created and used to highlight the state transition diagrams.

4.4 Examples

4.4.1 Conversation without Error

The first step in verifying conversations is to build the conversations. Figure 18 is an image of agentTool showing a system with two agents and a conversation between them.
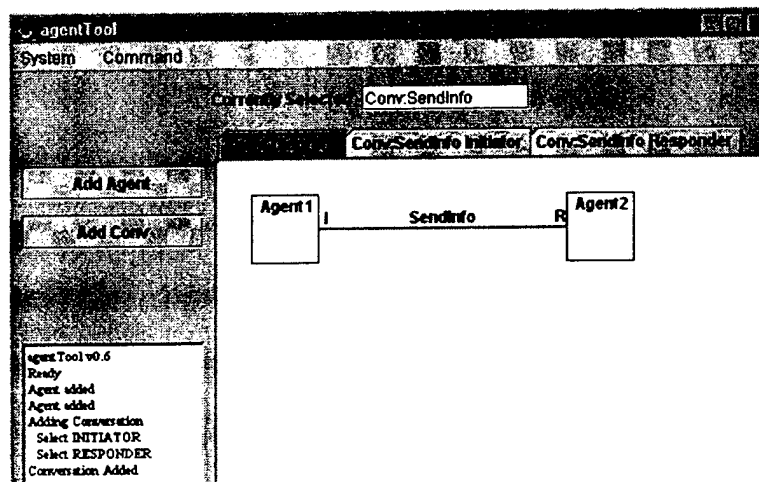

Figure 18: Conversation between Agents

The only properties of the conversation a user can define from this screen is the name of the conversation (*SendInfo*) and the direction of the conversation (who is the *initiator* and who is the *responder*). By clicking on the conversation line, two tabs appear. One tab is to define the state diagram for the initiator and one tab for the responder.

By clicking on one of the two tabs, a new window appears that automatically provides a start and an end state for the conversation designer. It is assumed every conversation has a start and an end state. In this window, designers add states and transitions to create a complete state transition diagram. Figure 19 is an image of the *initiator* side of the *SendInfo* conversation.

The complementary side of this state transition diagram is the *responder* side of the *SendInfo*
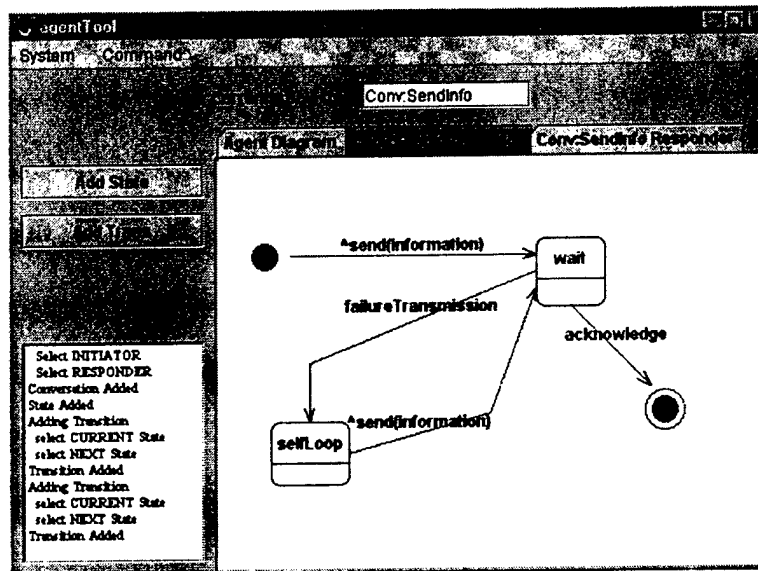
conversation and is shown in Figure 20.

Figure 19: Initiator Side of SendInfo Conversation

The *SendInfo* conversation is now completely specified and is ready to be verified. The

verification process is invoked by clicking on the Command pull down menu and choosing

Verify Conversations. A state table is created from the states and transitions of the state

transition diagram and this state table is used to create the Promela Code. Figure 21 is the

Promela Code created from this conversation.

The automated tool Spin is now invoked to check the syntax of the Promela code. If the

code is syntactically correct, Spin generates an analyzer to determine if protocol errors exist in the

conversation. The first check is for deadlocks. Spin determines if deadlocks exist by seeing if

either side of the conversation terminates while not having reached its end state. Spin calls this

kind of error an *invalid end-state* error. If the conversation is deadlocked, a message is displayed

in a text window that tells the user exactly where the deadlock occurred. The offending state

transition is also highlighted on the graphical state transition diagram. The highlighting can only

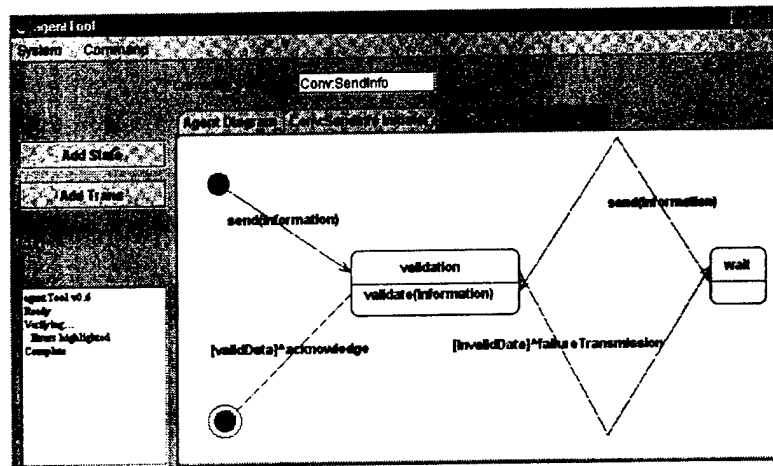be removed by re-verifying the conversation.

Figure 20: Responder Side of SendInfo Conversation

After the deadlock check is performed, the conversation is checked for livelock by

checking for states that are never entered. Spin calls this type of error a *non-progress* error

because the conversation has not made progress in these particular states. Again, if an error

condition exists, a message is displayed in the text window telling the user exactly where the non-

progress states are and the non-progress states are highlighted on the state transition diagrams.

They remain highlighted until the conversation is re-verified. If a deadlock condition is detected,

a trace file is created by Spin that allows a simulation be run that pinpoints the location of the

error. This message trace is also displayed in the text window to help the user to find the source

of the conversation errors. Figure 22 is the output messages displayed for the user when

verifying the SendInfo conversation.

Since there were no errors in this conversation, no error messages were displayed and no

states or transitions were highlighted. The next example will implement a conversation that has a

deadlock condition in it.

```
mtype = {send, acknowledge, failureTransmission, invalidData,
         validData };
chan SendInfo = [1] of {mtype};
proctype SendInfoInitiator()
{
   progressStartState:
      do
      :: SendInfo!send -> goto progresswait
      od;
   progresswait:
      do
      :: SendInfo?acknowledge -> goto progressEndState
      :: SendInfo?failureTransmission -> goto progressselfLoop
      od;
   progressselfLoop:
      do
      :: SendInfo!send -> goto progresswait
      od;
   progressEndState:
      do
      :: break
      od;
}
proctype SendInfoResponder()
{
   progressStartState:
      do
      :: SendInfo?send -> goto progressvalidation
      od;
   progressvalidation:
      do
      :: invalidData -> SendInfo!failureTransmission; goto
         progresswait
      :: validData -> SendInfo!acknowledge; goto progressEndState
      od;
   progresswait:
      do
      :: SendInfo?send -> goto progressvalidation
      od;
   progressEndState:
      do
      :: break
      od;}
init{
   atomic
   {
      run SendInfoInitiator();
      run SendInfoResponder()
   }
}
```

Figure 21: Promela Code of SendInfo Conversation

```
!!!!!!!!!! OUTPUT OF SPIN ANALYSIS !!!!!!!!!!

Analysis Completed... Evaluating Analysis...

********** OUTPUT FROM DEADLOCK CHECK **********

      CONVERSATION IS NOT DEADLOCKED!!!

********** OUTPUT FROM PROGRESS CHECK **********

      CONVERSATION DOES NOT HAVE UNUSED STATES!!!

********** OUTPUT FROM SIMULATED RUN **********

No trace available
```

Figure 22: Output From SendInfo Verification Run

4.4.2 Conversation with Error

The conversations shown thus far are error free. However, agentTool provides excellent user feedback when errors are detected. In order to demonstrate agentTool's error detecting and reporting capability, a new conversation must be created between two agents. Figure 23 shows the new conversation and agent added to the previous example.
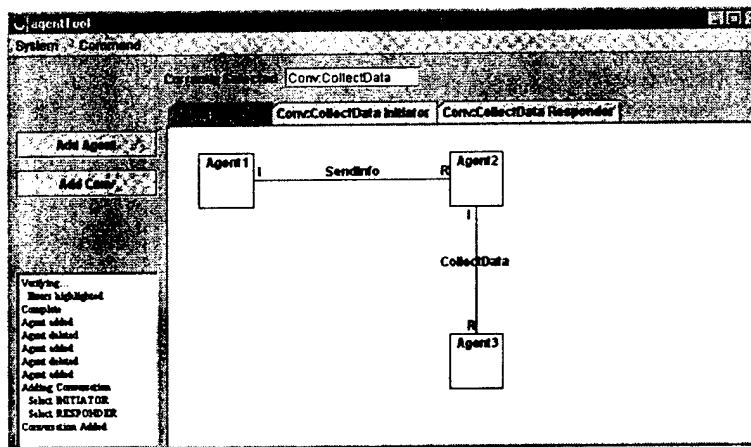

Figure 23: Two Conversations with Three Agents

The *CollectData* conversation must now be described. As before, there is an *initiator* side and a *responder* side to the conversation. Figure 24 shows the *initiator* side of the conversation while Figure 25 shows the *responder* side of the conversation.
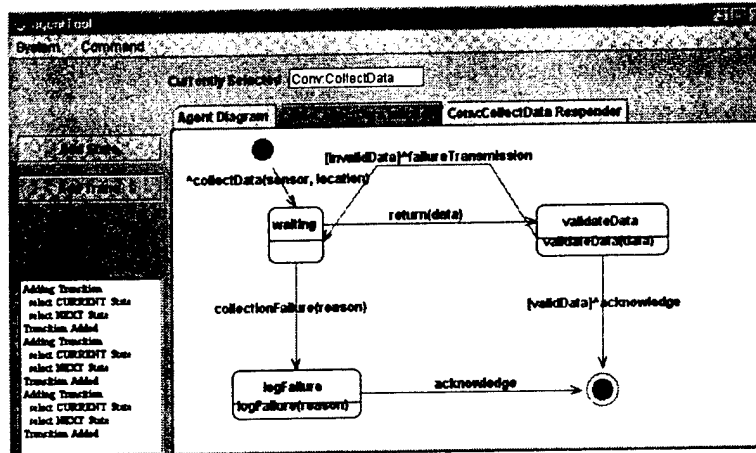
Figure 24: Initiator Side of CollectData Conversation

The initiator side of the CollectData conversation has an error in it. The transition from

the `logFailure` state that is labeled `acknowledge` is incorrect. As drawn, the transition is

waiting to receive an `acknowledge` message before transitioning to the end state. The transition

should be drawn so that it automatically sends an `acknowledge` message when in the `wait` state

and then immediately transitions to the `end` state. This incorrectly labeled transition will cause

the *CollectData* conversation to be deadlocked. Figure 26 is the Promela code for the collect data
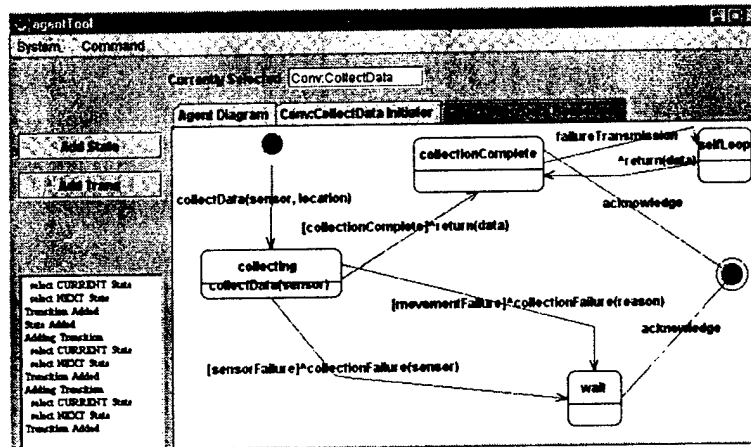
conversation.

Figure 25: Responder Side of CollectData Conversation

```
 mtype = { validData, invalidData, failureTransmission, acknowledgecollectData,return,
collectionFailure, sensorFailure, movemtFailure, collectionComplete };

chan CollectData = [1] of {mtype};

proctype CollectDataInitiator()
{
    progressStartState:
        do
        :: CollectData!collectData> goto progresswaiting
        od;
    progresswaiting:
        do
        :: CollectData?return> goto progressvalidateData
        :: CollectData?collectionFailure goto progresslogFailure
        od;
    progressvalidateData
        do
        :: invalidData-> CollectData!failureTransmission; goto progresswaiting
        :: validData-> CollectData!acknowledge; goto progressEndState
        od;
    progresslogFailure:
        do
        :: CollectData?acknowledge> goto progressEndStat
        od;
    progressEndState:
        do
        :: break
        od;
}

proctype CollectDataResponder()
{
    progressStartState:
        do
        :: CollectData?collectData> goto progresscollecting
        od;
    progresscollecting:
        do
        :: sensorFalure -> CollectData!collectionFailure; goto progresswait
        :: movementFailure> CollectData!collectionFailure; goto progresswait
        :: collectionComplete> CollectData!return; goto progresscollectionComplete
        od;
    progresscollectionComplete
        do
        :: CollectData?acknowledge> goto progressEndState
        :: CollectData?failureTransmission goto progressselfLoop
        od;
    progresswait:
        do
        :: CollectData?acknowledge> goto progressEndState
        od;
    progressselfLop:
        do
        :: CollectData!return> goto progresscollectionComplete
        od;
    progressEndState:
        do
        :: break
        od;
}

init
{
    atomic
    {
        run CollectDataInitiator()
        run CollectDataResponder()
    }
}
```

Figure 26: Promela Source Code for CollectData Conversation

When the user verifies these two conversations, a message window appears that gives the

status of the verification. As soon as the error is detected, the color of the text in the window

Figure 28: Highlighted Transition from CollectData Conversation

```
DEADLOCK CONDITION EXISTS IN THE FOLLOWING CONVERSATION:
Conversation Name = CollectData
Participant Name = Responder
Current State = wait
State Transition = acknowledge

DEADLOCK CONDITION EXISTS IN THE FOLLOWING CONVERSATION:
Conversation Name = CollectData
Participant Name = Initiator
Current State = logFailure
State Transition = acknowledge
```

Figure 29: Deadlock Messages from Message Window

4.4.3 Message Sequence Verification

Once conversations are defined and verified, specific message sequences that traverse the conversations can also be verified. Currently agentTool does not have the capability to graphically represent message sequence charts. However, message sequence charts can be represented via message sequence tables. Message sequence tables are very similar to state tables except the state information is not required. All that is needed is the *conversation* the message is a part of, the *initiator* and the *responder* of the message, and of course the *message*. Figure 30 is a message sequence chart that can be verified using the above two conversations.

Figure 31 shows a message sequence table for the message sequence chart of Figure 30. Before the message sequence can be verified, the conversations must be valid. Therefore, the

CollectData conversation must be corrected by changing the received acknowledge message

from the logFailure state to a transmitted acknowledge message from the logFailure state.
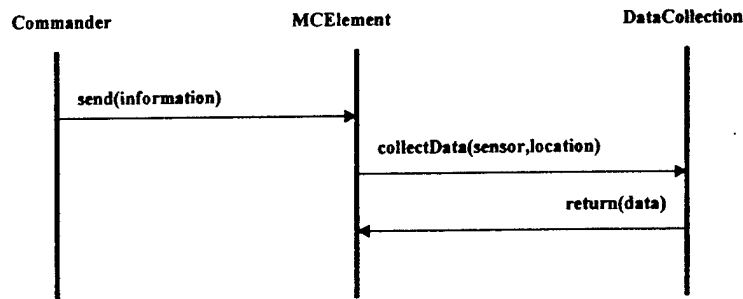
Figure 30: Message Sequence Chart for SendInfo and CollectData Conversations

```
SendInfo;Initiator;Responder;send
CollectData;Initiator;Responder;collectData
CollectData;Responder;Initiator;return
SendInfo;Initiator;Responder;send
```

Figure 31: Message Sequence Table for SendInfo and CollectData Conversations

As described in Section 3.4.5, a message sequence is verified by making a never claim

that states the desired sequence can never occur. Spin then tries to detect the message sequence,

and if it finds the sequence a never claim violation is raised. This is a very efficient way to find

a message sequence using a state space analyzer. Appendix B shows the message window output

after searching for the message sequence in Figure 31. If the message sequence is valid, a trace

of the messages is provided to show how the sequence was found.

If the trace is not valid, Spin will not be able to find the never claim. Depending on the

machine's capabilities, verifying a message sequence does not exist may take quite a bit of time.

Figure 32 is a message sequence table that contains an invalid message sequence. The send

message in the CollectData conversation is invalid.

```
SendInfo;Initiator;Responder;send
CollectData;Initiator;Responder;send
CollectData;Responder;Initiator;return
SendInfo;Initiator;Responder;send
```

Figure 32: Invalid Message Sequence Table

The results of the invalid message sequence verification are displayed in the message window and are referenced in Figure 33.

```
PLEASE STAND BY... TESTING MESSAGE SEQUENCE...

     !!!!!!!!!! OUTPUT OF SPIN ANALYSIS !!!!!!!!!!

     Analysis Completed... Evaluating Analysis...

     ********* OUTPUT FROM MESSAGE SEQUENCE CHECK *********

          MESSAGE SEQUENCE IS INVALID!!!

     ********* SEQUENCE TRACE IS AS FOLLOWS *********

     Message Sequence Invalid... - No trace available

********* TESTING COMPLETED *********
```

Figure 33: Invalid Message Sequence Output

Since the message sequence is invalid, no trace exists.

4.5 Analysis

Spin can check for many types of errors (Holzmann, 1997). However, agentTool does not currently provide the capability to check for all of them. This section will discuss what can and cannot currently be detected.

4.5.1 Errors Detected

4.5.1.1 Conversation Deadlocks

Conversation deadlocks are detected if there are no intervening factors such as hardware failures or timing problems. This is accomplished by performing an exhaustive state space search for deadlock conditions.

Figure 34 shows a conversation with a deadlocked condition. The transitions causing the deadlock are highlighted. The transition on the initiator side of the conversation is incorrect in

that it should be labeled as *transmitting* an `acknowledge` message instead of *receiving* an `acknowledge` message.
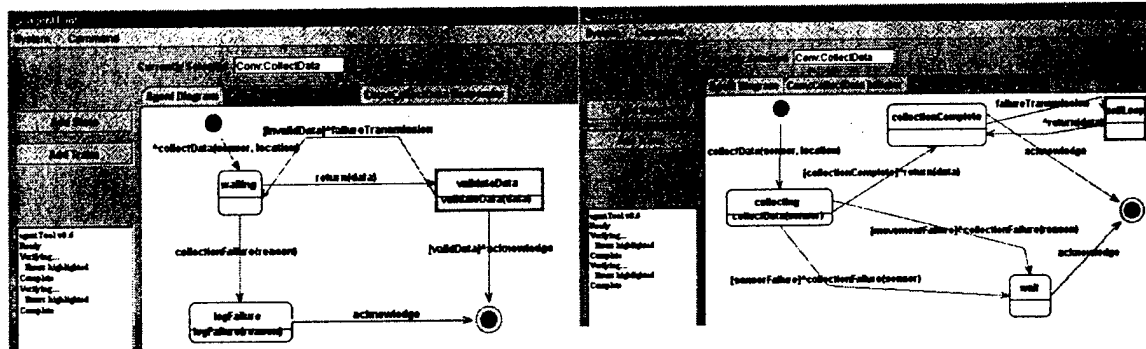
Figure 34: Conversation with Deadlock Condition Detected

4.5.1.2 Unused States

Unused states are detected by checking for non-progress loops. If a state is not used, it is not entered into and a non-progress error is generated.

Figure 35 shows a conversation with an unused state. The transition leading to the unused state (`State2`) is never enabled. The transition is waiting for a received message (`c`) that never is sent by the other side of the conversation. Therefore, the state can never be entered into and is highlighted to assist the system designer.
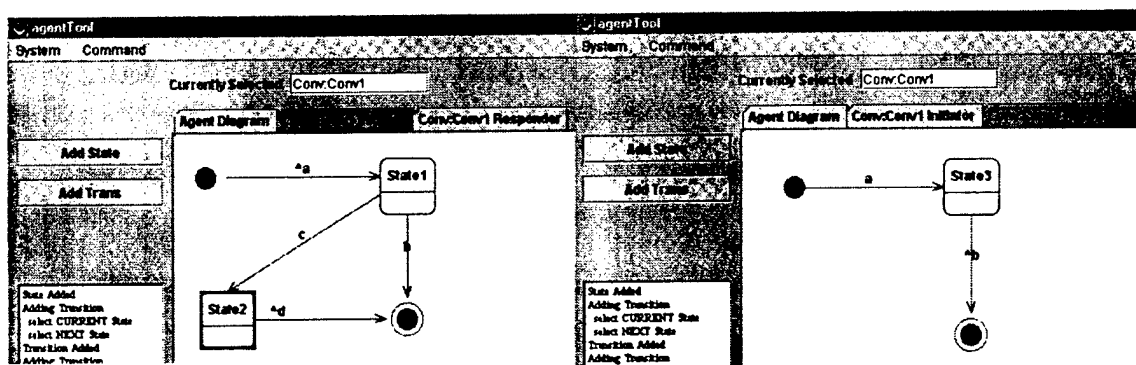
Figure 35: Conversation with Unused State Detected

4.5.1.3 Unused Messages

Unused messages are detected when they are not taken off the message channel, thereby leaving messages on the buffer. Since messages placed on the channel must be matched by a receiving process that takes them off the buffer, any unused messages will generate deadlock errors. This might not actually be a deadlock condition, but the error raised will generate enough information for the user to identify the source of the problem.

Figure 36 shows a conversation with an unused message. The transition from State1 has a transmitted message (b) that is not received by the other half of the conversation, thus causing a blockage.
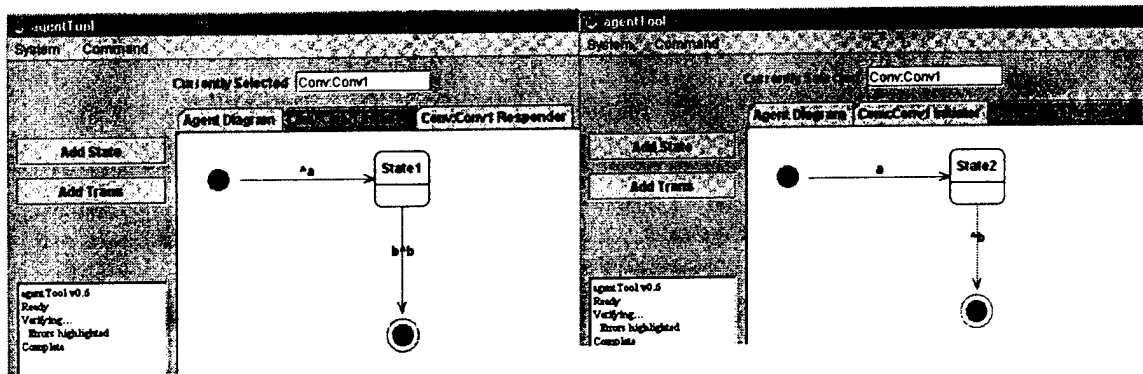
Figure 36: Conversation with Unused Message Detected

4.5.1.4 Mislabeled Transitions

Mislabeled transitions are detected when Spin is first run. If the syntax is incorrect, Spin cannot compile the Promela code into the executable analyzer. Feedback is provided via a message window when a syntax error occurs. Figure 37 shows the error messages generated when an invalid character (?) is used in a transition.

64

AFIT/GCS/ENG/00M-12

4.5.1.5 Inability to Create Required Sequences

Inability to create required sequences is detected using never claims. The desired

message sequence is modeled using a never claim, and if Spin does not generate a never claim

violation, the message sequence does not exist. Section 4.4.3 describes how an invalid message

sequence is detected and Figure 33 shows the messages after detecting the message sequence

does not exist.

```
spin: line   1 "verify", Error: syntax error saw 'operator: ?'
spin: line   9 "verify", Error: syntax error saw 'operator: ?'
spin: line  25 "verify", Error: undeclared variable: a    saw       ';'
    near 'goto'
spin: line  29 "verify", Error: undeclared variable: b    saw       ';'
    near 'goto'
spin: line  41 "verify", Error: proctype Conv1Initiator not found

1 mtype = { ?a, a, b };
2
3 chan Conv1 = [0] of {mtype};
4
5 proctype Conv1Initiator()
6 {
7 progressStartState:
8        do
9        :: ?a -> Conv1!a; goto progressState1
10              od;
```
Figure 37: Conversation Error Messages from Mislabeled Transition

4.5.2 Undetectable Errors

There are some communication errors that agentTool and Spin cannot currently detect.

These errors would be difficult for any automated system to detect; however, they are mentioned

here for completeness. There are plans to implement a *syntax checker* in agentTool that will

detect many of these errors such as state transition diagrams and guard conditions that are

incorrectly specified.

65

4.5.2.1 Timing Errors

Timing errors caused by system properties cannot be detected by Spin. The

conversations may be valid, but if a system property causes a conversation to pause indefinitely,

the complementary conversation is deadlocked until the system property allows the conversation

to continue. In this scenario, the conversations are valid and have been verified. Nevertheless,

the overall system will not perform correctly.

Figure 38 shows a conversation that is valid and verified. However, one of the transitions

(initiator side from start state) contains a guard condition that, if it never becomes true, will

prevent the conversation from completing.
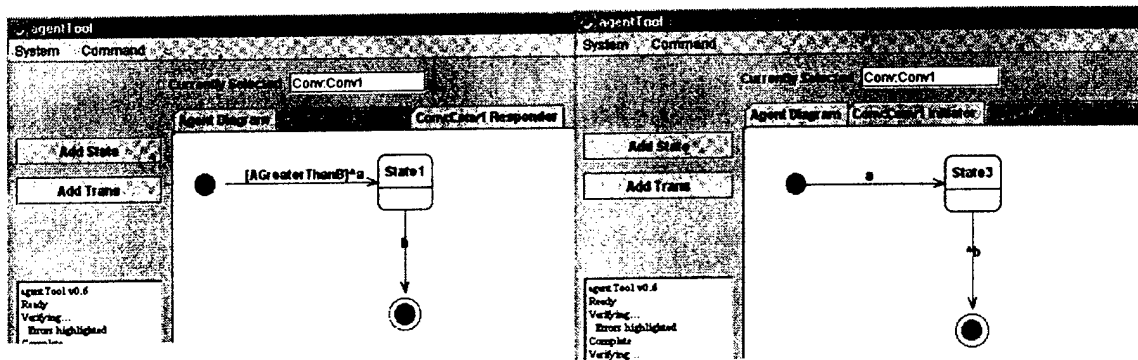
Figure 38: Timing Error Not Detected in Conversation

4.5.2.2 Floating States

Floating states (states with no transitions) cannot be detected by agentTool and Spin

because they are not passed from agentTool via a state table to the verifier. If a state does not

have any transitions, it is not included in the state table and it is non-existent as far as the verifier

is concerned. Figure 39 shows a state diagram created with agentTool that contains a floating

state. The conversation is valid and the floating state is ignored.

66

4.5.2.3 Hardware Failures

Hardware failures that cause infinite conversation loops cannot be detected by agentTool and Spin. The conversations are valid and have been verified, but if a sensor or other piece of hardware continues to send the same message in the context of a valid conversation, the conversation can become livelocked and the conversation cannot progress.
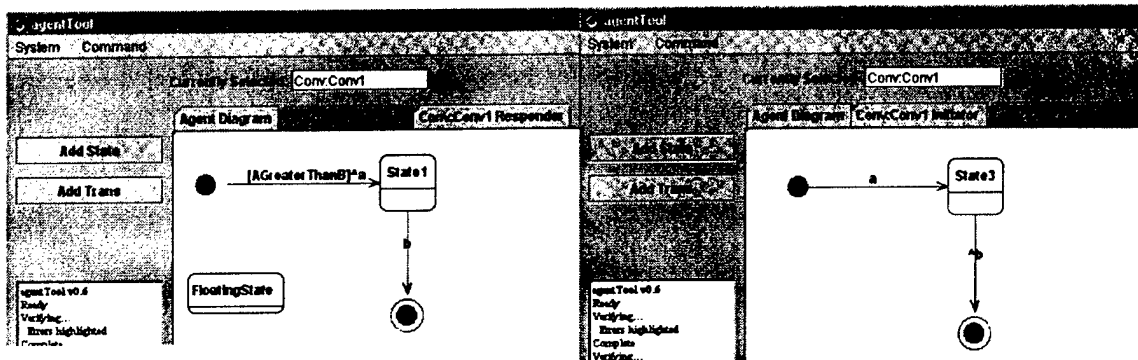
Figure 39: Floating State in Conversation

4.5.2.4 Guard Conditions

Guard conditions whose logic is specified incorrectly cannot be detected by agentTool and Spin. If a guard condition is specified as part of a conversation, agentTool uses a figurative representation of the guard condition to verify the conversation. If the guard condition consists of an algebraic formula that is written incorrectly, Spin will never know. Figure 40 shows a conversation with a guard condition specified incorrectly. The logic is wrong (A>5 && A<5).

4.5.2.5 Interacting Conversations Deadlock

Interacting conversations deadlock that results when two conversations are contending for a common resource cannot be detected by agentTool and Spin. Even though the conversations are valid, they can deadlock waiting for the same resource. Figure 41 shows a conversation where both sides are waiting on the same file, but neither can have access to it.
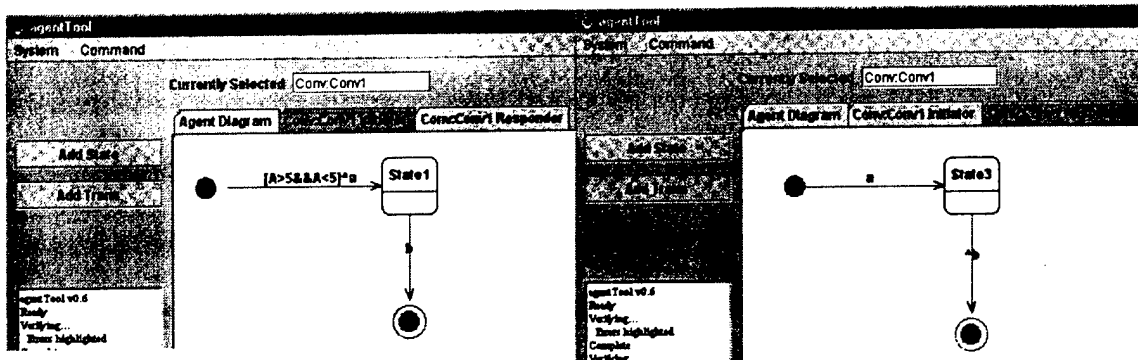
Figure 40: Incorrectly Specified Guard Condition in Conversation
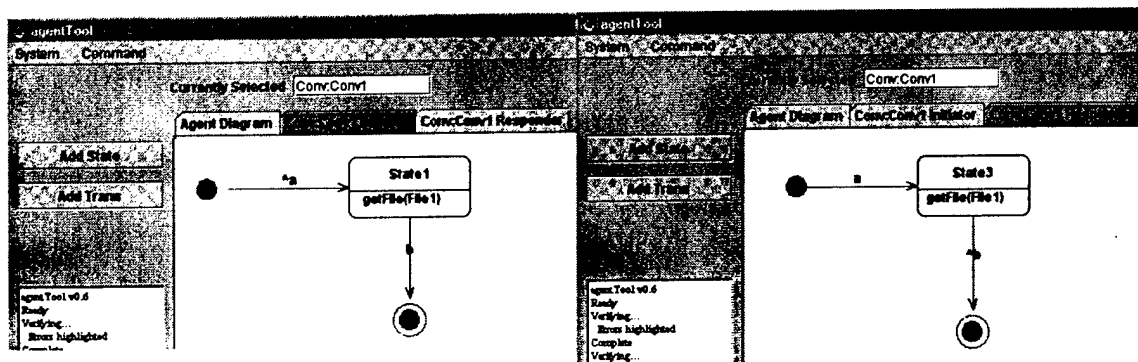
Figure 41: Interacting Conversations Deadlock

4.6 Summary

This chapter has demonstrated how agent conversations can be verified in agentTool using Promela and Spin. The input is via graphical state transition diagrams while the feedback to the user is both graphical and textual. Many critical communication centric errors are detected by agentTool and Spin. However, not all errors are detected by the automated tool so the final burden rests on the user to ensure the newly created multi-agent system is tested sufficiently.

V. Conclusions and Future Work

5.1 Introduction

The previous chapters of this thesis demonstrated how the conversations in a multi-agent system could be automatically verified. This chapter summarizes the conclusions from the previous chapters, and suggests areas of future work that will enhance and extend this research.

5.2 Conclusions

The previous chapters presented a methodology for automatically verifying multi-agent conversations and a prototype implementing this methodology. The following sections present conclusions obtained from this research.

5.2.1 Automatic Verification of Multi-agent Conversations

The automatic verification of conversations is a five step process that takes a graphical representation of a conversation via a state transition diagram, converts the state transition diagram to a state table, and creates a formal representation from the state table which can be formally verified. Creation of the state transition diagrams and state tables are straightforward. Creation of the formal representation requires in-depth knowledge of the formal language used. Spin is an excellent modeling language because it is designed to represent communication protocols. Other formal languages may be used, such as Communicating Sequential Processes or Calculus of Communicating Systems, but these languages are very difficult to understand and adapt to agent conversations.

This methodology is appropriate for verifying conversations in a closed agent system, where agents communicate through known and predictable state-based conversations. This methodology can also verify message sequences exist given a set of conversations. Figure 42 is a

state transition diagram used to graphically define an agent's conversation. Figure 43 is a message sequence chart used to illustrate the possible sequence of messages involving potentially many agent conversations.

The following is a summary of the types of errors detected through this methodology:

- Conversation deadlocks are detected if there are no intervening factors such as hardware failures or timing problems.

- Unused states are detected by checking for non-progress loops.

- Unused messages are detected when they are not taken off the message channel, thereby leaving messages on the message buffer.

- Mislabeled transitions are detected when Spin is first run by executing a syntax checker provided by Spin.

- The inability to create required sequences is detected using never claims.
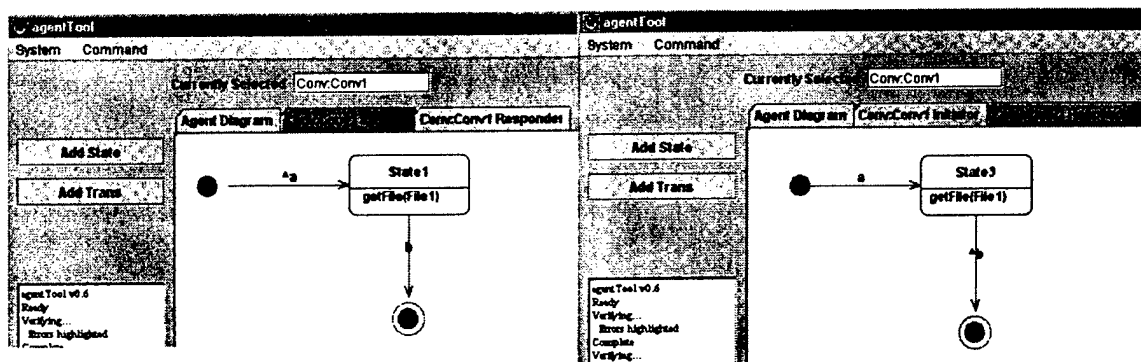
Figure 42: State Transition Diagram

A few errors cannot be detected at this time using this methodology. The following is a brief list summarizing undetectable errors:

- Timing errors caused by system properties will cause valid conversations to hang-up.

- Floating states (states with no transitions) cannot be detected because they are not passed from the graphical interface to the verifier.

- Hardware failures that cause infinite loops cannot be detected.

- Guard conditions incorrectly specified cannot be detected because the verifier does not evaluate guard conditions.

- Interacting conversations deadlock that results when two conversations are contending for a common resource cannot be detected.
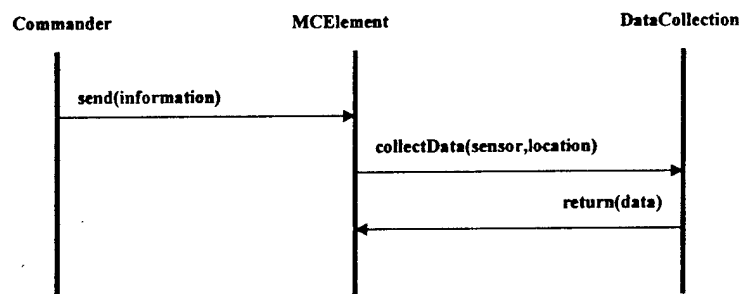
Figure 43: Message Sequence Chart

5.2.2 Implementation with agentTool

With agentTool, agent conversations are modeled using state transition diagrams. These state transition diagrams are automatically converted into Promela source code and verified with Spin. Feedback is provided to the system designer through text windows and graphical highlighting of error conditions in the original state transition diagrams. Although agentTool is still in development, it is a valuable tool for assisting the multi-agent system developer in building complex systems.

5.3 Future Work

AgentTool verification can be made more complete by adding a syntax checker to catch typographical and logical errors before attempting to verify conversations. Since agents designed

with agentTool are state-based, their designs should also be able to be verified using Promela and Spin.

5.3.1 Development of a Syntax Checker

Programming language compilers such as C and JAVA contain a syntax checker to ensure the programs written in their language are specified correctly. The syntax of agent and conversation specifications made with agentTool should also be evaluated by a checker to ensure they are written correctly. A syntax checker for agentTool would ensure 1) invalid characters (such as !?@#) are not used in conversation specifications, 2) guard conditions are logically correct, and 3) "do" actions required in conversation states or transitions are implemented in the agent's behavior. A syntax checker would also perform such tasks as ensuring at least one message (transmit, receive, or both) is associated with a transition.

The Object Constraint Language (OCL), developed by Integrated Business Engineering Language, IBM, is part of the Unified Modeling Language from version 1.1 on (Rational, 1997). OCL is based on standard set theory and is used to specify invariants on classes and types in the class model, to describe pre- and post conditions on operations and methods, and to describe guards. OCL can be used to write expressions that evaluate to true or false, thus making it a good choice for defining relational algebra formulas.

IBM has written a parser for OCL that can perform some basic syntax checking. This parser may be incorporated into the agentTool architecture and used to verify specifications written in OCL are correct. Portions of agent and conversation specifications in agentTool are written in OCL and should be verifiable with an OCL parser.

5.3.2 Verification of an Agent's State-based Behavior

Since an agent's behavior can be defined using state transition diagrams (Robinson, 2000), a system of agents can be verified by simulating the response agents have when receiving and sending messages through conversations with other agents. Figure 44 shows an agent's state based interior.
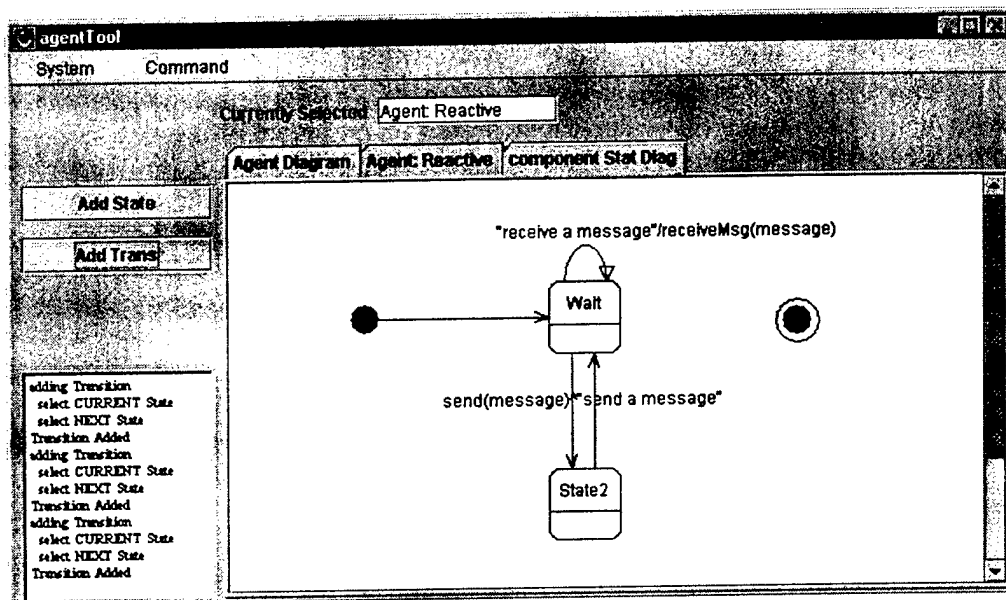
Figure 44: Agent State Based Interior (Robinson, 2000)

This would be similar to the research performed in this thesis and many of the same techniques reapplied.

5.4 Summary

This research addresses a critical need in the development of multi-agent systems, automatic verification. Automatic verification brings together the skills of computer scientists and mathematicians resulting in software that is more dependable and robust than previously attainable with traditional software development tools. Software engineers no longer have to hope their agent conversations will work as expected. Automatic verification, once thought

impossible to accomplish, is attainable and provides a much-needed tool for multi-agent

development systems.

BIBLIOGRAPHY

Cleaveland, Rance. "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems," ACM Transactions on Programming Languages and Systems, 15(1): 36-72 (January 1993).

DeLoach, Scott A. "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems," Proceedings of a Workshop on Agent-Oriented Information Systems (AOIS '99). 45-57. Seattle WA. May 1, 1999.

Hailpern, Brent T. Verifying Concurrent Processes Using Temporal Logic. New York: Springer-Verlag, 1982.

Harel, David. "Statecharts: A Visual Formalism For Complex Systems," Science of Computer Programming, Volume 8: 231-274 (1987).

Hinchey, M.G. and J.P. Bowen. High-Integrity System Specification and Design. London: Springer-Verlag, 1999.

Hoare, C.A. Communicating Sequential Processes. New York: Prentice-Hall, 1985.

Holzmann, Gerard J. "The Model Checker Spin," IEEE Transactions On Software Engineering, Volume 23, Number 5: 279-295 (May 1997).

Kelley, Jay W. Air Force 2025. 2025 Support Office, Air University, Air Education and Training Command. Air University Press, August 1996.

Lowe, Gavin and Bill Roscoe. "Using CSP to Detect Errors in the TMN Protocol," IEEE Transactions on Software Engineering, Vol. 23, No. 10: (October 1997).

Manna, Zohar and Amir Pnueli. The Temporal Logic of Reactive and Concurrent Systems. New York: Springer-Verlag, 1992.

Milner, Robin. Communication and Concurrency. New York: Prentice-Hall, 1989.

Pressman, Roger S. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 1997.

Raphael, Marc J. Knowledge Base Support For Design and Synthesis of Multi-agent Systems. MS thesis, AFIT/ENG/00M-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

Rational Software Corporation. Object Constraint Language Specification. version 1.1, 1 September 1997.

Robinson, David J. A Component Based Approach to Agent Specification. MS thesis, AFIT/ENG/00M-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

Shalikashvili, John M. Joint Vision 2010. Joint Staff: Pentagon, 1999.

Stevens, Perdita. The Edinburgh Concurrency Workbench. User Manual. University of Edinburgh, November 1998.

Sycara, Katia P. "Multiagent Systems," American Association for Artificial Intelligence: 79-92, (Summer 1998).

Wood, Mark F. Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. MS thesis, AFIT/ENG/00M-26. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

APPENDIX A: MESSAGES FOR ERROR CONVERSATION

```
PLEASE STAND BY... TESTING CONVERSATIONS...

   !!!!!!!!!! OUTPUT OF SPIN ANALYSIS !!!!!!!!!!

   Analysis Completed... Evaluating Analysis...

   ********** OUTPUT FROM DEADLOCK CHECK **********

         CONVERSATION IS DEADLOCKED!!!

   ********** OUTPUT FROM PROGRESS CHECK **********

         CONVERSATION DOES NOT HAVE UNUSED STATES!!!

   ********** OUTPUT FROM SIMULATED RUN **********

   proc 0 = :init:
   proc 1 = SendInfoInitiator
   proc 2 = SendInfoResponder
   proc 3 = CollectDataInitiator
   proc 4 = CollectDataResponder
   q\p   0   1   2   3   4
      1   .   .   .     CollectData!collectData
      1   .   .   .   .   CollectData?collectData
      1   .   .   .   .   CollectData!collectionFailure
      1   .   .   .   CollectData?collectionFailure
      2   .   SendInfo!send
      2   .   .   SendInfo?send
      2   .   .   SendInfo!acknowledge
      2   .   SendInfo?acknowledge
   spin: trail ends after 16 steps
   -------------
   final state:
   -------------
   #processes: 5
                 queue 2 (SendInfo):
                 queue 1 (CollectData):
      16:   proc  4 (CollectDataResponder) line  92 "verify" (state 27)
      16:   proc  3 (CollectDataInitiator) line  65 "verify" (state 24)
      16:   proc  2 (SendInfoResponder) line   46 "verify" (state 24) <valid
            endstate>
      16:   proc  1 (SendInfoInitiator) line   25 "verify" (state 22) <valid
            endstate>
      16:   proc  0 (:init:) line 114 "verify" (state 6) <valid endstate>
   5 processes created

   ********** DETAILED DEADLOCK INFORMATION **********

         DEADLOCK CONDITION EXISTS IN THE FOLLOWING CONVERSATION:
         Conversation Name = CollectData
         Participant Name = Responder
         Current State = wait
         State Transition = acknowledge

         DEADLOCK CONDITION EXISTS IN THE FOLLOWING CONVERSATION:
         Conversation Name = CollectData
         Participant Name = Initiator
         Current State = logFailure
         State Transition = acknowledge
```

********* TESTING COMPLETED *********

Appendix B: Messages from Message Sequence Verification

PLEASE STAND BY... TESTING MESSAGE SEQUENCE...

!!!!!!!!!!! OUTPUT OF SPIN ANALYSIS !!!!!!!!!!!

Analysis Completed... Evaluating Analysis...

********** OUTPUT FROM MESSAGE SEQUENCE CHECK **********

       MESSAGE SEQUENCE IS VALID!!!

********** SEQUENCE TRACE IS AS FOLLOWS **********

```
proc 0 = :init:
proc 1 = SendInfoInitiator
proc 2 = SendInfoResponder
proc 3 = CollectDataInitiator
proc 4 = CollectDataResponder
proc 5 = CollectDataResponder
proc 6 = CollectDataInitiator
proc 7 = SendInfoInitiator
proc 8 = SendInfoResponder
q\p  0   1   2.  3   4   5   6   7   8
  1  .   .   .   .   .   .   .   SendInfo!send
  1  .   .   .   .   .   .   .   .   SendInfo?send
  1  .   .   .   .   .   .   .   .   SendInfo!failureTransmission
  1  .   .   .   .   .   .   .   SendInfo?failureTransmission
  1  .   .   .   .   .   .   .   SendInfo!send
  2  .   .   .   .   .   .   CollectData!collectData
  1  .   .   .   .   .   .   .   .   SendInfo?send
  1  .   .   .   .   .   .   .   .   SendInfo!failureTransmission
  1  .   .   .   .   .   .   .   SendInfo?failureTransmission
  1  .   .   .   .   .   .   .   SendInfo!send
  2  .   .   .   .   .   CollectData?collectData
  1  .   .   .   .   .   .   .   .   SendInfo?send
  1  .   .   .   .   .   .   .   .   SendInfo!failureTransmission
  1  .   .   .   .   .   .   .   SendInfo?failureTransmission
  2  .   .   .   .   .   CollectData!collectionFailure
  1  .   .   .   .   .   .   .   SendInfo!send
  1  .   .   .   .   .   .   .   .   SendInfo?send
  1  .   .   .   .   .   .   .   .   SendInfo!failureTransmission
  2  .   .   .   .   .   .   CollectData?collectionFailure
  1  .   .   .   .   .   .   .   SendInfo?failureTransmission
  1  .   .   .   .   .   .   .   SendInfo!send
  1  .   .   .   .   .   .   .   .   SendInfo?send
  2  .   .   .   CollectData!collectData
  1  .   .   .   .   .   .   .   .   SendInfo!failureTransmission
  1  .   .   .   .   .   .   .   SendInfo?failureTransmission
  1  .   .   .   .   .   .   .   SendInfo!send
  1  .   .   .   .   .   .   .   .   SendInfo?send
  2  .   .   .   .   CollectData?collectData
  2  .   .   .   .   CollectData!return
  2  .   .   .   CollectData?return
  2  .   .   .   CollectData!failureTransmission
  2  .   .   .   .   CollectData?failureTransmission
  2  .   .   .   .   CollectData!return
  1  .   SendInfo!send
  2  .   .   .   CollectData?return
```

```
spin: trail ends after 98 steps
-------------
final state:
-------------
#processes: 10
                queue 1 (SendInfo): [send]
                queue 2 (CollectData):
  98:    proc  8 (SendInfoResponder) line  15 "verify" (state 10)
  98:    proc  7 (SendInfoInitiator) line  34 "verify" (state 11)
  98:    proc  6 (CollectDataInitiator) line  56 "verify" (state 15)
  98:    proc  5 (CollectDataResponder) line  88 "verify" (state 28)
  98:    proc  4 (CollectDataResponder) line  83 "verify" (state 23)
  98:    proc  3 (CollectDataInitiator) line  60 "verify" (state 24)
  98:    proc  2 (SendInfoResponder) line   9 "verify" (state 3)
  98:    proc  1 (SendInfoInitiator) line  34 "verify" (state 11)
  98:    proc  0 (:init:) line 110 "verify" (state 10) <valid endstate>
  98:    proc  - (:never:) line 136 "verify" (state 26) <valid endstate>
10 processes created
```

********** TESTING COMPLETED **********

VITA

Captain Timothy H. Lacey was born on 26 October 1961 in Thomasville, Georgia. He graduated from Colquitt County High School in Moultrie, Georgia in June 1979. He entered the Air Force's enlisted force January 1983 and completed his undergraduate studies with a Bachelor of Science degree in Computer Science, magna cum laude, in June 1991. He was commissioned through the Air Force's Officer Training School (OTS) in November 1992.

Captain Lacey's first assignment was at Mountain Home AFB, Idaho as a Russian simulator operator and maintainer in October 1983. In December 1989, he was assigned as a computer programmer to Scott AFB, Illinois. While at Scott AFB, he completed his undergraduate degree and received his commission through OTS. His first assignment as an officer was to Hill AFB, Utah in April 1993. There he was a programmer manager for the Air Force Mission Support System used by pilots to preplan their flights. In November 1995 he was assigned to Bolling AFB, DC and worked for the Defense Intelligence Agencies Engineering Review Board. In July 1998, he entered the Graduate School of Engineering's Computer Systems Engineering program, Air Force Institute of Technology. Upon graduation, he will be assigned to AFIT/SC at Wright Patterson AFB where he will be in charge of the Information Systems Branch (SCB).

REPORT DOCUMENTATION PAGE		*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE A FORMAL METHODOLOGY AND TECHNIQUE FOR VERIFYING COMMUNICATION PROTOCOLS IN A MULTI-AGENT ENVIRONMENT			**5. FUNDING NUMBERS**
6. AUTHOR(S) Timothy H. Lacey, Captain, USAF			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765			**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/00M-12
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NM Attn: Captain Freeman Alex Kilpatrick 801North Randolph Street Room 732 9-65 Arlington VA 22203-1977 Commercial: (703) 696-6565			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
Maj Scott A. DeLoach, ENG, DSN:785-3636, ext. 4622

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

As network bandwidth increases, distributed applications are becoming increasingly prevalent. Systems using these applications are very complicated to build and must be dependable. Software agents are ideal for breaking complicated problems into manageable subtasks. Agent conversations, a series of messages passed between agents, are the cornerstone of multi-agent systems and must be deemed correct before being placed into service. The purpose of this research was to develop a formal methodology and technique to verify that the communication protocols defined in a multi-agent environment were valid. This was accomplished by examining agent conversations before deploying the system. An additional goal of this research was to develop a proof-of-concept module for agentTool that automatically verified some of the important properties identified in this methodology.

In the end, this research produced a methodology for automatically verifying conversations and the methodology was implemented in the agentTool software development environment. Improvements and future work was also recommended as a result of this effort.

14. SUBJECT TERMS Agents, Conversations, Formal Verification, State Transition Diagrams, Promela, Spin, agentTool, Deadlock, Livelock, Infinite Loops, Multi-agent Environment, Automatic Verification, Automatic, Formal, Message Sequences, Feedback, Errors, State Tables			15. NUMBER OF PAGES 93
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL